IN THE UNITED STATES DISTRICT COURT
FOR THE SOUTHERN DISTRICT OF NEW YORK

06 CV 13565

JUDGE ROBINSON

**INTERNATIONAL BUSINESS MACHINES CORPORATION,**

**Plaintiff,**

**-vs.-**

**PLATFORM SOLUTIONS, INC.,**

**Defendant.**

**Civil Action No. _____**

**JURY TRIAL DEMANDED**



## COMPLAINT

Plaintiff International Business Machines Corporation ("IBM"), by and through its attorneys Quinn Emanuel Urquhart Oliver & Hedges, LLP, as and for its Complaint against defendant Platform Solutions, Inc. ("PSI"), states as follows:

### Nature of the Action

1. This is an action for breach of contract, patent infringement, and a declaratory judgment by IBM against PSI.

2. IBM has invested substantial amounts of time, effort, know-how, creativity, and money to develop its computer systems, the architectures for those systems, and the operating systems and other software programs that are compatible with and run on those architectures. IBM's efforts have been directed to developing combinations of computer hardware and software specifically tailored to meet the most demanding customer requirements. As a result of IBM's investment, IBM's computer systems and programs provide unparalleled performance, reliability, availability,

serviceability, and security and have been widely accepted for use by customers in environments where accuracy, data integrity, and reliability are critically important.

3. PSI seeks to usurp the value of IBM's investment. PSI has developed and is bringing to market and offering for sale computer systems ("emulator systems") that seek to imitate IBM's computers and that PSI claims will run IBM's copyrighted operating systems and other software programs on computers other than the ones for which the IBM software was written.

4. PSI is using, promoting, and offering for sale PSI's emulator systems in New York and elsewhere and is asserting that (a) PSI's emulator systems allow users to translate and run IBM's operating systems on computers other than the ones for which the IBM software was written and allow users to obtain the patented functionality of IBM's computer architectures on computers that do not implement those architectures; and (b) purchasers of PSI's emulator systems will be able to license IBM's operating systems and other software from IBM for use on PSI's emulator systems.

5. In developing, operating, and promoting its emulator systems, PSI has breached the contracts under which IBM has licensed PSI to use IBM software. IBM licenses certain of its operating systems and other software to end user customers, including PSI, pursuant to the IBM Customer Agreement ("ICA"). PSI has obtained licenses to use IBM software pursuant to an ICA that expressly prohibits, among other things, any translation of the licensed software programs. In violation of this express prohibition, to which PSI contractually bound itself in obtaining the applicable licenses, PSI has developed emulator systems that, according to PSI, translate IBM software that runs on IBM computer architectures so as -- again according to PSI -- to allow that

software to be run on computer architectures other than the ones for which the software was written. By using IBM's software in conjunction with PSI's emulator systems and thereby translating the IBM software, PSI has breached its software license agreements with IBM. IBM seeks damages and a declaration that it is authorized to terminate PSI's software licenses based on PSI's breaches of the terms of the ICA.

6. In addition, in developing, operating, and promoting its systems, PSI has used IBM's own intellectual property without authorization. PSI's emulator systems, to work as PSI claims, necessarily infringe IBM patents. PSI's marketing program is a blatant attempt to infringe on IBM's intellectual property and to convert to PSI the fruits of IBM's substantial investments in developing computer systems, architectures, operating systems, and other software. By this action, IBM seeks to preclude PSI from marketing and offering for sale emulator systems that infringe IBM's patents. IBM seeks relief under the patent laws in order to prevent irreparable harm to IBM from PSI's infringing activities, including its sales of, and/or offers to sell, directly and through third parties, infringing emulator systems.

7. As PSI has breached IBM's software license agreements and infringed IBM's patents in developing, operating, and promoting its infringing emulator systems, PSI has insisted that IBM agree to license IBM's patents to PSI and IBM's copyrighted operating systems and other software for use on PSI's systems. IBM has declined to grant the requested licenses because, among other reasons, PSI is infringing IBM's patents. PSI has responded by threatening baseless antitrust litigation seeking substantial alleged damages. IBM seeks a declaration that its refusal to license IBM's patents to PSI and IBM's copyrighted operating systems and other software for use on PSI's emulators

does not violate the antitrust laws -- a declaration for which IBM relies, in part, on the same evidence of patent infringement by PSI that forms the basis for IBM's claims for affirmative relief under the patent laws.

**The Parties**

8. Plaintiff IBM is a corporation organized and existing under the laws of New York, having its principal place of business at New Orchard Road, Armonk, New York 10504. IBM's business activities, including research and development, manufacturing, marketing, and service, are primarily in the field of information processing products and services. IBM develops, manufactures, markets, and services computers, computer equipment, and software on a worldwide basis in competition with a large number of firms both inside and outside of the United States.

9. Defendant PSI is a corporation organized and existing under the laws of California, having its principal place of business at 501 Macara Avenue, Suite 101, Sunnyvale, California 94085.

**Jurisdiction and Venue**

10. IBM's claims arise under the patent and antitrust laws of the United States, 35 U.S.C. §§ 1 *et seq.* and 15 U.S.C. §§ 1 *et seq.* This Court has jurisdiction over the subject matter of this action pursuant to 28 U.S.C. §§ 1331 and 1338(a).

11. This Court has jurisdiction over IBM's breach of contract claim pursuant to 28 U.S.C. §§ 1332 and 1367. There is complete diversity of citizenship, and the amount in controversy exceeds $75,000, exclusive of interest and costs.

12. This Court has authority to grant declaratory relief pursuant to the Declaratory Judgment Act, 28 U.S.C. §§ 2201 *et seq.*

13. This Court has personal jurisdiction over PSI because PSI (directly and through its licensees and marketing partners): (a) is transacting business in the State of New York; (b) has offered its infringing products for sale in this District; (c) has an Eastern Regional Manager located in Rochester, New York; (d) has raised substantial funding for its emulator development program in New York; (e) has licensed IBM software pursuant to license agreements that are governed by New York law and were negotiated on the basis of correspondence and communications directed to New York; (f) has placed orders through IBM's New York website for IBM software which PSI has used in its emulator development and marketing program; and (g) has held meetings with IBM in New York, and directed correspondence to IBM in New York, concerning PSI's emulator systems in which it has requested that IBM license its patents to PSI and its copyrighted operating systems and other software for use on PSI emulator systems and threatened antitrust litigation. Because any use of IBM's software in conjunction with PSI's emulator systems involves the translation of IBM's software in violation of the applicable license agreements, all demonstrations and uses of PSI's emulator systems in New York involve breaches of contract that have occurred in New York. In addition, PSI has committed (and permitted its licensees and marketing partners to commit) acts of patent infringement and/or has contributed to or induced acts of patent infringement by others in New York, and has taken actions elsewhere directed to selling PSI emulator systems to customers based in New York; and/or PSI expects or should reasonably expect its acts to have consequences in New York and the acts have caused and will cause injury to IBM in New York. Among other things, PSI has publicly presented its infringing emulator systems in New York, including at a SHARE Conference in New York;

maintains an interactive website accessible to New York residents and contacts New York residents who express an interest in PSI's emulator systems from the office of its Eastern Regional Manager in Rochester, New York; and has met with customers in New York to promote the sale of, and offer to sell, PSI's emulator systems. An infringing PSI emulator system has been installed and used at a customer location in New York.

14. Venue is proper in this judicial district pursuant to 28 U.S.C. §§ 1391(b), 1391(c), and 1400(b). A substantial part of the events giving rise to IBM's request for declaratory relief occurred in this judicial district. IBM and PSI met at IBM's offices in New York to discuss IBM's decision not to license its patent and software, and PSI directed correspondence to IBM's offices in New York, requesting a license for IBM's patents and software and threatening antitrust litigation. In addition, the actions of PSI (directly and through its licensees and marketing partners) in New York and directed to New York that infringe IBM's patents and breach IBM's software license agreements are sufficient to give rise to personal jurisdiction in New York and accordingly support venue in this District. These activities include publicly presenting PSI's infringing emulator system at an industry conference in New York; using IBM software in New York in breach of the applicable license agreements; meeting with customers in New York to promote the sale of, and offer to sell, PSI emulator systems; and installing an infringing PSI emulator system at a customer location in New York. Alternatively, the doctrine of pendent venue applies to IBM's patent infringement claims, so that this Court may in its discretion hear those claims, which arise out of the same nucleus of operative facts as IBM's declaratory judgment claim.

## Factual Background

### A. IBM Has Invested Heavily In Computer Systems, Architectures, Operating Systems, And Other Software.

15. For over forty years, IBM has invested substantial time, effort, know-how, and creativity, and substantial amounts of money, in developing and improving its computer architectures and the computer systems that implement those architectures.

16. As a result of its investments over time, IBM has developed System z. System z is the brand name for IBM's current mainframe computer systems. System z has evolved from IBM computer systems dating back to 1964. Its predecessors include IBM's System/390® ("S/390®"), which was introduced in 1990. System z is an umbrella term for: IBM zSeries® servers (introduced in 2000), IBM System z9 servers (introduced in 2005), and IBM operating systems and other IBM software that run on zSeries® or z9 servers.

17. A computer's architecture defines the logical structure and functional operation of the computer. System z computers implement IBM's current 64-bit z/Architecture®. As a result of IBM's investments, IBM's z/Architecture® has evolved over time from predecessor architectures, including IBM's 31-bit Enterprise Systems Architecture/390® ("ESA/390"), Enterprise Systems Architecture/370 ("ESA/370"), and several earlier architectures.

18. Operating systems comprise the fundamental software that controls the execution of programs on the computer and provides basic services such as resource allocation, scheduling, input/output control, and data management. IBM's operating systems, like its architectures and computer systems, are the product of substantial investments over time.

19. Particular operating systems are designed to run on computers that implement a particular architecture and to capitalize on the features and characteristics of that architecture. IBM's copyrighted OS/390® operating system, for example, was designed to run on IBM's S/390® computers, which implement IBM's ESA/390 Architecture. IBM's copyrighted z/OS® operating system is the successor operating system to OS/390® and is designed to run on IBM's System z computers, which implement IBM's z/Architecture®. The relationship between IBM's computer architectures and the operating systems designed to run on those architectures is one of the important factors contributing to the accuracy and reliability of IBM's computer systems and to customer acceptance of those systems for mission-critical applications.

20. In addition to mainframe operating systems, architectures, and computers that implement those architectures, IBM has invested substantial time, effort, know-how, creativity, and money in developing other software programs that work in conjunction with those operating systems and computers and themselves capitalize on the features and characteristics of IBM's computer architectures. Examples of such other IBM software programs include IBM's Customer Information Control System ("CICS®") and IBM's Database 2 ("DB2®"). Like IBM's operating systems, these programs are designed to operate in conjunction with IBM's computer architectures and to capitalize on the features and characteristics of IBM's architectures.

21. IBM holds a substantial portfolio of patents relating to System z and predecessor computer systems. IBM's patents are directed, among other things, to aspects of its z/Architecture®, as well as to aspects of the predecessor ESA/390 Architecture. IBM has further sought to protect its substantial investment in computer

intellectual property by copyrighting its mainframe operating systems and other software and by imposing reasonable contractual restrictions on the manner in which customers may use those computer programs.

**B. PSI Now Seeks To Convert IBM's Substantial Investment In Intellectual Property By Developing And Marketing Systems That Emulate IBM's Computer Architectures.**

22. Recognizing the substantial value of IBM's intellectual property, PSI has developed and is now implementing a business model that seeks to usurp the value of IBM's investment in mainframe computer systems, architectures, operating systems, and other IBM software programs. PSI seeks to do this by emulating, or mimicking, IBM's computer architectures. An emulator is a combination of software, firmware, and/or hardware added to a computer that implements one architecture (*e.g.*, the Intel Architecture) for the purpose of translating computer programs written for a different architecture and enabling those programs to be run on the computer to which the emulator has been added. The purpose of an emulator is to allow the computer to which it is added to accept the same data and the same instructions, run the same programs, and achieve the same results as does the computer whose architecture is being emulated. According to PSI, PSI's emulator systems accomplish this objective by translating IBM's copyrighted software into a set of instructions that can be executed by a processor that is not capable of executing the original IBM instructions.

23. PSI claims to be "the first developer of a new generation of mainframe computers compatible with the broadest set of datacenter environments and operating systems, including IBM® z/OS®." PSI asserts that its emulator systems are compatible with IBM's ESA/390 Architecture and z/Architecture® computer systems. The PSI emulator systems run on an Intel® Itanium®-based server. According to PSI, its emulator

systems are capable of running IBM's OS/390® and z/OS® operating systems and other IBM computer programs that run on those operating systems, such as IBM's CICS® and DB2®.

24. As PSI's public statements recognize, a complete emulator of an IBM computer architecture must, by definition, fully and exactly mimic the relevant IBM architecture. IBM's z/Architecture® is defined in an approximately 1000-page Principles of Operation ("POP"), the fifth edition of which was published in 2005. If the POP indicates that a facility is present, and PSI's emulator does not exactly mimic it, software that attempts to make use of the facility will not work properly. To be a viable emulator, PSI's emulator system would have to be able to accurately run z/OS® and at least any additional System z software required by the customers that PSI is targeting.

25. PSI has asserted that its emulator systems are able to execute "the 1200+ instructions from the z/OS and S/390 instruction set," and that its emulator systems are compatible with -- *i.e.*, capable of running -- IBM's OS/390® and z/OS® operating systems, other IBM software intended to run on OS/390® and z/OS®, and vendor and customer application software intended to run on those operating systems. PSI (through one of its licensees and marketing partners) has recently elaborated on this statement, asserting that PSI's emulator systems will run IBM's "latest" z/OS® operating system. PSI has further asserted that z/OS® workloads will run "identically" on PSI's emulator systems as on an IBM mainframe computer.

26. zSeries® servers and their predecessors have been the backbone of commercial computing for decades, renowned for their reliability, scalability, availability, serviceability, and other industrial-strength attributes. The zSeries® server

and its z/OS® operating system were designed for environments requiring very high performance, reliability, accuracy, and security. Many z/OS® customers have business requirements for continuous system availability. System down time or unplanned outages, even of short duration, can cost millions of dollars in lost revenue or other significant negative business impact. IBM has a strong interest in ensuring that z/OS® is not used on computer systems with which z/OS® is not fully compatible or used in ways that have the potential to undermine either the reputation of z/OS® for accuracy, data integrity, and reliability or customer acceptance of z/OS® for mission-critical applications.

### C. In Developing And Promoting Its Emulator Systems, PSI Had Breached Its Software License Agreements With IBM.

27. In March 2004, PSI executed an ICA with IBM. A true and correct copy of PSI's ICA is attached hereto as Exhibit 1. IBM has consistently made clear to PSI that the granting of software licenses has not in any way granted PSI a patent license or any express or implied rights, licenses, or immunities under any IBM patents or other intellectual property. The ICA, by its terms, grants PSI "only the licenses and rights specified. No other licenses or rights (including licenses or rights under patents) are granted." The ICA expressly prohibits PSI from, among other things, "translating" licensed ICA Programs, including z/OS®. After executing the ICA, PSI licensed copies of z/OS® pursuant to the terms of the ICA.

28. PSI's emulator systems use software, which PSI refers to as firmware, to mimic IBM's ESA/390 Architecture and z/Architecture® by translating IBM software written for computer systems using those architectures into instructions that can be executed by computer systems incorporating a different architecture. As described by

PSI, z/OS® (and any other zSeries® software to be run on the emulator system) runs on top of the PSI "firmware" layer. In this arrangement, each instruction in z/OS® (and any other zSeries® software) that is being executed on the PSI emulator system is translated into one or more Itanium® instructions by the PSI firmware. PSI's translated Itanium® instructions are then executed by the Itanium® processor, with the intent of producing the same result as if the z/OS® (or zSeries® software) instructions translated by the PSI firmware were instead executed on an IBM zSeries® server.

29. According to PSI, PSI's emulator systems translate what PSI calls "legacy instructions" contained in IBM's copyrighted software into what PSI calls "translated instructions." PSI's emulator systems use what PSI calls dynamic just-in-time translation, in which the object code of IBM's operating systems is translated, the translated instructions are stored or "cached" in the memory of the computer system on which the PSI emulator system is installed, and the translated instructions yielded by the emulator are executed by that computer system.

30. As described in U.S. Patent No. 7,092,869 ("the '869 patent") issued to Ronald N. Hilton, PSI's founder and Chief Technology Officer, "[e]ach particular legacy instruction is translated into one or more particular translated instructions for emulating the particular translated legacy instruction." As further described in the '869 patent, the so-called legacy instructions at issue are for a "legacy system" (such as an IBM computer system having an ESA/390 Architecture or z/Architecture®); "the legacy instructions are object code instructions compiled/assembled for the S/390 [or System z] system and the translated instructions are for execution in a [different] architecture." As further described in the '869 patent, PSI's emulators use "translated code for emulating" the IBM

computer system; the "native executable code" in the copyrighted IBM computer programs licensed by PSI under the ICA is "processed by the emulator . . . to produce translated code . . . for execution by the target system . . . according to an architecture different from the native architecture." As described in the '869 patent, moreover, the "legacy code translator" in PSI's emulator systems stores detailed information about the translation in translation store" and the "translated code . . . output from this cache . . . is executed" by the target computer system on which the emulator system is running. The net result, according to the '869 patent, is a "computer-implemented method for dynamic emulation of legacy instructions comprising," among other things, (a) "accessing said legacy instructions"; (b) "for each particular legacy instruction, translating the particular legacy instruction into one or more particular translated instructions for emulating the particular legacy instruction"; and (c) ultimately "executing the translated instructions to emulate execution of the legacy instructions."

31. PSI's public presentations on its emulator systems have indicated that those systems operate in the manner described in the '869 patent, and that their purpose and function is to translate IBM's object code and to store the object code translations for execution on a computer using an Itanium® processor. Based on PSI's public descriptions, PSI's emulator systems translate z/OS® into instructions that can be executed by the Itanium® processor in the computers on which the emulators are installed. Even if the resulting translations were accurate, and if the translated instructions in fact allowed an Itanium® processor to execute the translated instructions in a manner that provided performance comparable to the execution of the original IBM instructions on a zSeries® server, such translation of z/OS® is prohibited by the ICA.

**D. PSI's Emulator Systems Infringe IBM Patents.**

32. Based on the purpose and nature of PSI's emulator systems, as stated by PSI, and IBM's knowledge of emulator technology, the making, using, selling, or offering for sale of those systems necessarily infringes IBM patents, and/or will contribute to or induce infringement of those patents by users of PSI's emulator systems. IBM's conclusion that PSI's emulator systems infringe IBM's patents is not based on an examination of one of PSI's systems because IBM's request that it be permitted to conduct an examination of a PSI emulator system and its offer to discuss infringement in greater detail following such an examination have been ignored by PSI, which has instead threatened antitrust liability as a result of IBM's rejection of PSI's demands that IBM license its patents to PSI and its copyrighted software for use on PSI's emulator systems.

33. The IBM patents that are infringed by PSI include the following:

a. On December 9, 1997, the United States Patent and Trademark Office ("USPTO") issued U.S. Patent No. 5,696,709 entitled "Program Controlled Rounding Modes" (hereinafter "the '709 patent"). A true and correct copy of the '709 patent is attached hereto as Exhibit 2.

b. On October 20, 1998, the USPTO issued U.S. Patent No. 5,825,678 entitled "Method And Apparatus For Determining Floating Point Data Class" (hereinafter "the '678 patent"). A true and correct copy of the '678 patent is attached hereto as Exhibit 3.

c. On September 14, 1999, the USPTO issued U.S. Patent No. 5,953,520 entitled "Address Translation Buffer For Data Processing System Emulation Mode" (hereinafter "the '520 patent"). A true and correct copy of the '520 patent is attached hereto as Exhibit 4.

d. On November 16, 1999, the USPTO issued U.S. Patent No. 5,987,495 entitled "Method and Apparatus For Fully Restoring A Program Context Following An Interrupt" (hereinafter "the '495 patent"). A true and correct copy of the '495 patent is attached hereto as Exhibit 5.

e. On October 5, 2004, the USPTO issued U.S. Patent No. 6,801,993 entitled "Table Offset For Shortening Translation Tables From Their Beginnings" (hereinafter "the '993 patent"). A true and correct copy of the '993 patent is attached hereto as Exhibit 6.

34. IBM is the owner of all right, title, and interest in and to the '709, '678, '520, '495, and '993 patents by assignment, with full and exclusive right to bring suit to enforce each of these patents, including the right to recover for past infringement.

35. IBM may identify additional IBM patents infringed by PSI's emulator systems after IBM has an opportunity to examine a PSI emulator system.

**E. IBM Has Declined To License Its Patents To PSI And Its Copyrighted Software For Use On PSI's Emulator Systems, And PSI Has Threatened Antitrust Litigation.**

36. PSI and IBM have met and corresponded concerning PSI's activities. IBM has advised PSI that PSI's emulator systems infringe various IBM patents, and has offered to give PSI opportunities to demonstrate that its emulator systems are non-infringing if PSI so believes. PSI has declined IBM's invitations and requests for information about, and access to, PSI's emulator systems.

37. IBM and PSI have met and corresponded concerning PSI's demands that IBM agree to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems. IBM has declined to provide the patent and software licenses demanded by PSI. PSI has asserted, in words or substance, that IBM's

refusal to license IBM's patents to PSI and IBM's software for use on PSI's emulator systems is somehow unlawful and damaging to PSI. As early as March 16, 2001, PSI stated in correspondence to IBM that its "major concern" was "IBM's stated decision not to license the z/Architecture at all at this point," and that PSI assumed IBM would "continue to reevaluate that position, given the potential anti-trust issues that could be raised." On October 29, 2002, PSI wrote to IBM and (a) asserted that IBM's decision not to license its software for use on PSI's systems is "not consistent with the long term IBM practice of licensing its software regardless of the implementation of the computing platform"; (b) requested licensing terms for commercial operation of IBM's software on PSI's emulator systems; and (c) stated that "each day that goes by is directly impacting our development schedule." On December 23, 2002, at a time when PSI said it was preparing "for the commercial introduction of our product line early next year," PSI wrote to IBM's President and CEO (a) criticizing IBM's decision not to license IBM's software for use on PSI's systems; (b) arguing that IBM's refusal conflicts with historical "precedent" and is "discriminatory" and "purely arbitrary"; and (c) claiming that the effect of IBM's decision is "intentionally anti-competitive" and that "our survival as a company has been placed in immediate jeopardy as a result." On February 7, 2003, PSI wrote to IBM seeking a letter of intent from IBM confirming IBM's willingness to license its software for use on PSI's emulator systems and asserting that "[t]he unexpected uncertainty on this point has severely hampered the execution of our business plans, jeopardizing the entire venture." On November 10, 2003, PSI wrote to IBM and stated that IBM's licensing position with respect to z/OS® "has severely impacted our ability to deliver a 64 bit machine" and requested reconsideration of that position. On October 5,

2005, PSI wrote to IBM (a) again criticizing IBM's refusal to license IBM's software to be run on PSI's emulator systems; (b) arguing that IBM's licensing position is inconsistent with IBM's "prior practices and precedents"; and (c) asserting that IBM's licensing position is "causing confusion -- both to us and to our end user customers" and "is not just causing confusion in the market" but "is causing harm to our business."

38. At a meeting in New York in February 2006, PSI requested that IBM reconsider PSI's request that IBM grant PSI a patent license for PSI's emulator systems and agree to license z/OS® and other IBM software for use on PSI's emulator systems. On May 24, 2006, IBM declined PSI's request. IBM stated at that time that "IBM continues to believe that PSI's products infringe IBM's intellectual property rights" and that "we continue to see indications that PSI is engaged in infringing activity . . . . IBM has clearly articulated to PSI its belief that by developing and/or offering for sale a product that can run IBM's z/OS operating system, PSI is infringing a number of IBM patents, including IBM's z/Architecture patents. A non-exhaustive list of IBM U.S. patents potentially infringed by PSI was provided to you on August 18, 2005. To date, PSI has provided IBM with no substantive response. Instead, PSI has chosen to continue its development and marketing efforts notwithstanding IBM's rights and interests."

39. On June 8, 2006, PSI directed correspondence to IBM at IBM's offices in New York asserting that IBM's decision not to license its patents to PSI and not to license z/OS® to run on PSI systems was "completely unjustified." PSI asserted that IBM's position "will undoubtedly result in significant harm to both PSI and its customers" and "strongly urge[d]" IBM to reconsider its decision. In light of the history of communications between the parties, IBM reasonably construes this letter as threatening

antitrust litigation if IBM continues to decline to license its patents to PSI and its operating systems and other software for use on PSI's emulator systems.

40. On August 3, 2006, IBM declined PSI's request for reconsideration of its decisions not to grant PSI a patent license and not to license z/OS® and other software to run on PSI's emulator systems: "As we have explained, we believe that a PSI emulator that runs IBM's z/OS operating system infringes a number of IBM patents. We have repeatedly expressed this view to PSI, and you have acknowledged that PSI believes it requires patent licenses from IBM. In asking us to reconsider our decision, PSI has provided no new information. IBM would welcome the opportunity to examine one of PSI's systems and, following such an examination, would be willing to discuss PSI's infringements in greater detail." At the same time, IBM advised PSI that, "We are very concerned that, despite the fact that PSI is unlicensed to IBM's patents and has been informed that IBM will not license z/OS on PSI systems, PSI continues to make public statements that it intends to offer systems that run z/OS. IBM is extremely concerned that these statements will induce potential users of PSI systems to infringe IBM's intellectual property rights. Please ensure that PSI does not in any way represent or imply that PSI systems are authorized or eligible for a license to run the IBM z/OS operating system."

41. On August 9, 2006, PSI directed correspondence to IBM at IBM's offices in New York. PSI asserted that "PSI does not believe its systems infringe any patents that IBM may hold" in the fields of z/Architecture® and coupling; admitted that PSI "has not undertaken the lengthy effort and expense of a detailed infringement analysis" with respect to other IBM patents; and stated that it "was most surprised and disappointed" by

IBM's position that it would not grant PSI a patent license. PSI further stated that, "PSI believes that IBM's current posture in dealing with PSI to be unwarranted and calculated to cause it substantial harm. It is for this reason that PSI urged IBM to reconsider its position, and does so again here." IBM reasonably construes this correspondence as threatening antitrust litigation if IBM continues to decline to license its patents to PSI and its operating systems and other software for use on PSI's emulator systems.

**F. PSI's Recent Activities Have Brought The Parties' Dispute To A Head.**

42. In March 2006, PSI announced that it was demonstrating and "delivering to customers today around the world" emulator systems that run IBM's z/OS® operating system. In March 2006, PSI also publicly announced that its z/Architecture® emulator system would be "generally available" in the second half of 2006. In June 2006, PSI publicly described its then-current activities as involving beta test customer placements and initial early shipment program shipments, as well as establishing a "direct and channel sales force," and "ISV relationships." In July 2006, PSI issued a press release stating that, "Later this year, PSI expects to deliver z/OS® compatible servers that use dual-core processor technology to address the performance requirements of more than 90 percent of the z/OS installed base." PSI has recently publicly demonstrated its infringing emulator system in, among other places, Baltimore, Maryland; Houston, Texas; and San Jose and San Francisco, California; identified beta customers; and stated that its emulator would be generally available in the fourth quarter of 2006. PSI has been actively marketing and offering for sale its emulator systems to potential customers, including potential customers in New York, and has publicly stated that a PSI system is installed and in use at a customer location in New York. IBM remains unwilling to license its

patents to PSI or its software for use on PSI's emulator systems. Accordingly, the parties' dispute is now ripe.

43. In addition to PSI's direct activities, beginning in November 2006, one of PSI's licensees and marketing partners launched a new website trumpeting the availability of one system based on PSI's emulator systems and the imminent availability of another PSI emulator system. Following this announcement, PSI's emulator systems are now being actively marketed and offered for sale to potential customers, including potential customers in New York.

44. PSI (directly and/or through its licensees and marketing partners) has expressly and impliedly advised potential customers that they will be able to license IBM's mainframe operating systems and other software for use on PSI's emulator systems and has falsely stated, among other things, that (a) PSI's systems "will run" the "latest" IBM operating systems; (b) IBM software will be available for licensing for use on PSI's systems; (c) IBM will license its operating systems for use on PSI's systems in a "business as usual" manner; (d) licensing of 64-bit software from IBM is available for PSI's systems but not for a competing emulator system; (e) PSI is in discussions with IBM concerning software pricing for PSI systems and PSI will take care of software licensing issues with IBM; (f) software pricing for z/OS® will be the same as the price of that software when licensed on certain IBM machines; (g) PSI's systems have what PSI describes as "advanced partitioning capabilities that allow customers to control z/OS-based software licensing fees by isolation of individual workloads or logical server"; (h) PSI's systems will involve the use of a "[r]educed Z image" and therefore "qualify" for lower IBM software licensing rates for z/OS® and other IBM software; and has further

stated that (i) their lawyers "are ready for anything" and are prepared to sue IBM over a refusal to license IBM software for use on PSI's systems or the imposition by IBM of higher licensing fees for software used on PSI's systems than for software licensed for use on allegedly comparable IBM mainframe systems. As PSI reasonably expected, these statements and threats have been communicated to IBM.

45. As a result of these and other activities, an article appeared in a trade publication on September 26, 2006 entitled "A Joint Assault on the Mainframe Hardware Market." The article, which was subsequently posted on PSI's website, described PSI as having a series of computers "that can load and run software written for the [IBM] System z9 and its antecedents" and that are compatible with "IBM's current 64-bit processor architecture." The article asserts that PSI has "rights to obtain IBM software licenses, and the legal know-how required to preserve and extend these rights," and suggests that with the "commercial marketing of PSI systems," IBM "will supply and support its full range of mainframe software products."

46. In addition, and also as a result of these and other activities, IBM has begun to receive customer communications suggesting confusion in the market with respect to customers' ability to use IBM software in conjunction with PSI's emulator systems.

**COUNT ONE**

**(Breach of Contract)**

47. IBM realleges and incorporates herein the allegations of paragraphs 1 through 46 of this Complaint as if fully set forth herein.

48. PSI has licensed copies of z/OS® and other IBM software pursuant to the terms of the ICA.

49. The ICA is, by its terms, governed by New York law.

50. IBM has fully performed all of its obligations under its license agreements with PSI.

51. The ICA expressly prohibits PSI from, among other things, "translating" licensed ICA Programs, including z/OS®.

52. According to PSI, PSI's emulator systems translate what PSI calls "legacy instructions" contained in z/OS® into what PSI calls "translated instructions." Based on PSI's public descriptions, PSI's emulator systems translate IBM's copyrighted software into instructions that can be executed by the Itanium® processor in the computers on which the emulators are installed.

53. PSI has used IBM's z/OS® operating system licensed pursuant to the terms of the ICA in conjunction with PSI's emulator systems, and PSI's emulator systems have translated z/OS® in violation of the express prohibitions of the ICA. By so doing, PSI has breached its license agreements with IBM.

54. As a result of PSI's activities, IBM has been damaged in an amount to be proved at trial.

55. As a result of PSI's breaches of its license agreements with IBM, IBM is entitled, pursuant to the terms of the ICA and New York law, to terminate the ICA and to terminate PSI's authorization to use the licensed software.

## COUNT TWO

## (Infringement of the '709 Patent)

56. IBM realleges and incorporates herein the allegations of paragraphs 1 through 55 of this Complaint as if fully set forth herein.

57. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe, literally and/or under the doctrine of equivalents, the '709 patent by practicing one or more claims in the '709 patent in the manufacture, use, offering for sale, and sale of PSI's emulator systems.

58. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe the '709 patent by contributing to or actively inducing the infringement by others of the '709 patent by providing PSI's emulator systems.

59. PSI has willfully infringed the '709 patent.

60. PSI's acts of infringement of the '709 patent will continue after the service of this Complaint unless enjoined by the Court.

61. As a result of PSI's infringement, IBM has suffered and will suffer damages.

62. IBM is entitled to recover from PSI the damages sustained by IBM as a result of PSI's wrongful acts in an amount subject to proof at trial.

63. Unless PSI is enjoined by this Court from continuing its infringement of the '709 patent, IBM will suffer additional irreparable harm and impairment of the value of its patent rights. Thus, IBM is entitled to an injunction against further infringement.

**COUNT THREE**

**(Infringement of the '678 Patent)**

64. IBM realleges and incorporates herein the allegations of paragraphs 1 through 63 of this Complaint as if fully set forth herein.

65. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe, literally and/or under the doctrine of equivalents, the '678 patent by practicing

one or more claims in the '678 patent in the manufacture, use, offering for sale, and sale of PSI's emulator systems.

66. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe the '678 patent by contributing to or actively inducing the infringement by others of the '678 patent by providing PSI's emulator systems.

67. PSI has willfully infringed the '678 patent.

68. PSI's acts of infringement of the '678 patent will continue after the service of this Complaint unless enjoined by the Court.

69. As a result of PSI's infringement, IBM has suffered and will suffer damages.

70. IBM is entitled to recover from PSI the damages sustained by IBM as a result of PSI's wrongful acts in an amount subject to proof at trial.

71. Unless PSI is enjoined by this Court from continuing its infringement of the '678 patent, IBM will suffer additional irreparable harm and impairment of the value of its patent rights. Thus, IBM is entitled to an injunction against further infringement.

## COUNT FOUR

## (Infringement of the '520 Patent)

72. IBM realleges and incorporates herein the allegations of paragraphs 1 through 71 of this Complaint as if fully set forth herein.

73. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe, literally and/or under the doctrine of equivalents, the '520 patent by practicing one or more claims in the '520 patent in the manufacture, use, offering for sale, and sale of PSI's emulator systems.

74. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe the '520 patent by contributing to or actively inducing the infringement by others of the '520 patent by providing PSI's emulator systems.

75. PSI has willfully infringed the '520 patent.

76. PSI's acts of infringement of the '520 patent will continue after the service of this Complaint unless enjoined by the Court.

77. As a result of PSI's infringement, IBM has suffered and will suffer damages.

78. IBM is entitled to recover from PSI the damages sustained by IBM as a result of PSI's wrongful acts in an amount subject to proof at trial.

79. Unless PSI is enjoined by this Court from continuing its infringement of the '520 patent, IBM will suffer additional irreparable harm and impairment of the value of its patent rights. Thus, IBM is entitled to an injunction against further infringement.

**COUNT FIVE**

**(Infringement of the '495 Patent)**

80. IBM realleges and incorporates herein the allegations of paragraphs 1 through 79 of this Complaint as if fully set forth herein.

81. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe, literally and/or under the doctrine of equivalents, the '495 patent by practicing one or more claims in the '495 patent in the manufacture, use, offering for sale, and sale of PSI's emulator systems.

82. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe the '495 patent by contributing to or actively inducing the infringement by others of the '495 patent by providing PSI's emulator systems.

83. PSI has willfully infringed the '495 patent.

84. PSI's acts of infringement of the '495 patent will continue after the service of this Complaint unless enjoined by the Court.

85. As a result of PSI's infringement, IBM has suffered and will suffer damages.

86. IBM is entitled to recover from PSI the damages sustained by IBM as a result of PSI's wrongful acts in an amount subject to proof at trial.

87. Unless PSI is enjoined by this Court from continuing its infringement of the '495 patent, IBM will suffer additional irreparable harm and impairment of the value of its patent rights. Thus, IBM is entitled to an injunction against further infringement.

**COUNT SIX**

**(Infringement of the '993 Patent)**

88. IBM realleges and incorporates herein the allegations of paragraphs 1 through 87 of this Complaint as if fully set forth herein.

89. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe, literally and/or under the doctrine of equivalents, the '993 patent by practicing one or more claims in the '993 patent in the manufacture, use, offering for sale, and sale of PSI's emulator systems.

90. In violation of 35 U.S.C. § 271, PSI has infringed and is continuing to infringe the '993 patent by contributing to or actively inducing the infringement by others of the '993 patent by providing PSI's emulator systems.

91. PSI has willfully infringed the '993 patent.

92. PSI's acts of infringement of the '993 patent will continue after the service of this Complaint unless enjoined by the Court.

93. As a result of PSI's infringement, IBM has suffered and will suffer damages.

94. IBM is entitled to recover from PSI the damages sustained by IBM as a result of PSI's wrongful acts in an amount subject to proof at trial.

95. Unless PSI is enjoined by this Court from continuing its infringement of the '993 patent, IBM will suffer additional irreparable harm and impairment of the value of its patent rights. Thus, IBM is entitled to an injunction against further infringement.

**COUNT SEVEN**

**(Declaratory Judgment)**

96. IBM realleges and incorporates herein the allegations of paragraphs 1 through 95 of this Complaint as if fully set forth herein.

97. There is a real and actual controversy between IBM and PSI concerning IBM's refusal to license its patents to PSI and its copyrighted mainframe software for use on PSI's emulator systems. IBM's refusal to license IBM's patents to PSI and IBM's copyrighted operating systems and other software for use on PSI's emulators does not violate the antitrust laws because, among other reasons, the antitrust laws recognize IBM's right, under the patent and copyright laws, to refuse to license its patents and

copyrights. PSI's emulator systems infringe IBM patents, and the antitrust law specifically recognizes a copyright holder's right to decline to license copyrighted software for use on a system that infringes the copyright holder's patents. Thus, this controversy requires resolution of substantial questions of patent law and involves the same evidence of patent infringement by PSI that forms the basis for IBM's claims for affirmative relief under the patent laws. In addition, IBM has a strong interest in ensuring that z/OS® is not used on computer systems with which z/OS® is not fully compatible or used in ways that have the potential to undermine either the reputation of z/OS® for accuracy, data integrity, and reliability or customer acceptance of z/OS® for mission-critical applications.

98. Since PSI's emulator systems first came to IBM's attention, IBM has concluded, based on their purpose and nature and on IBM's knowledge of emulator technology, that those systems will necessarily infringe IBM patents. PSI has not ameliorated IBM's reasonable, good faith concerns that PSI's emulator systems infringe IBM patents.

99. Nevertheless, PSI (directly and/or through its licensees and marketing partners) has (a) demanded that IBM license IBM's patents to PSI and IBM's copyrighted mainframe operating systems and other software for use on PSI's emulator systems; (b) expressly and implicitly asserted that a refusal by IBM to license its patents, operating systems, and other software is anti-competitive and in violation of the antitrust laws; (c) asserted to IBM customers that purchasers of PSI's emulator systems will be able to license IBM's copyrighted operating systems and other copyrighted IBM software for the same license prices as users of allegedly comparable IBM mainframe systems;

(d) acknowledged confusion in the market over this issue; (e) has raised the specter of substantial alleged harm to PSI from IBM's decision not to license its patents and copyrighted software; and (f) advised IBM customers (in ways that it reasonably expected would be communicated to IBM) that their lawyers "are ready for anything" and are prepared to sue IBM.

100. IBM has refused to agree to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems despite PSI's demands because, among other reasons, IBM has no obligation to do so, and IBM is unwilling to allow its software to be run on an emulator system that infringes IBM's patents. PSI's emulator systems infringe IBM patents, and IBM has so advised PSI. In response to IBM's stated concerns about patent infringement, PSI has asserted, without providing IBM with any supporting information or any opportunity to examine a PSI emulator system, that PSI's emulators do not infringe certain patents directed to aspects of z/Architecture® identified by IBM as likely to be infringed by PSI's emulators. In light of PSI's public statements concerning the purpose and capabilities of its emulators, and IBM's knowledge of emulator technology, PSI's unsupported assertion is wrong. PSI has provided no assurances whatsoever concerning a larger group of patents identified by IBM as potentially infringed by PSI's emulator systems and has instead requested that IBM license PSI under IBM's patent portfolio -- an action that IBM is unwilling to take. Nevertheless, PSI (directly and through its licensees and marketing partners) has used in various public forums and is marketing and offering for sale emulator systems that necessarily infringe IBM's patents. IBM and PSI therefore have a real and actual controversy concerning the issue of patent infringement by PSI's emulator systems.

101. IBM has told PSI that IBM will not license its patents to PSI or its operating systems and other software for use on PSI's emulator systems. PSI has asserted that IBM's refusal to license IBM's patents, operating systems, and other software is anti-competitive, violates the antitrust laws, and is causing confusion in the market and substantial damage to PSI's business. PSI is wrong. The antitrust laws do not restrict IBM's rights, under the patent and copyright laws, to refuse to license IBM's lawfully acquired patents and copyrights. Further, it is IBM's reasonable, good faith belief that PSI's emulator systems infringe IBM's patents. That belief, standing alone, is a well-recognized and legally sufficient basis for IBM's decision to decline to license its operating systems and other software, as the antitrust laws do not require IBM to license its copyrighted software for use on a computer system that infringes IBM's patents. Nevertheless, IBM has reasonably construed PSI's statements (and statements made by one of PSI's licensees and marketing partners) as portending inevitable and imminent litigation over the propriety of IBM's patent and software licensing decisions. Judicial resolution of the parties' disputes is now required.

102. Litigation over the propriety of IBM's licensing position under the antitrust laws is now inevitable and imminent, in light of, among other things, (a) PSI's demands that IBM agree to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems; (b) PSI's baseless assertions that IBM's decision not to license its patents and software is anti-competitive and in violation of the antitrust laws; (c) PSI's statements concerning the impact on PSI's business of IBM's refusal to license its patents to PSI and its operating systems and other software for use on PSI emulator systems; (d) PSI's recent requests that IBM reconsider

its refusal to license its operating systems and other software for use on PSI's emulator systems and IBM's decision not to do so; and (e) statements by PSI (directly and/or through its licensees and marketing partners) that their lawyers "are ready for anything" and are prepared to sue IBM over IBM's licensing decisions. IBM and PSI have a real and actual controversy over IBM's refusal to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems.

103. IBM is entitled to a declaratory judgment that IBM's refusal to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems is not anti-competitive and does not violate the antitrust laws.

**PRAYER FOR RELIEF**

WHEREFORE, IBM prays for the following relief:

a. That this Court declare that IBM is entitled to terminate the ICA and all software licenses previously granted to PSI under the terms of the ICA as a result of PSI's breaches of the terms of the ICA;

b. That PSI, its officers, agents, servants, employees, and those persons acting in active concert or in participation with it be enjoined from further infringement of the '709 patent, the '678 patent, the '520 patent, the '495 patent, and the '993 patent pursuant to 35 U.S.C. § 283;

c. That this Court declare that IBM's refusal to license its patents to PSI and its copyrighted operating systems and other software for use on PSI's emulator systems is not anti-competitive and does not violate the antitrust laws;

d. That PSI be ordered to pay damages adequate to compensate IBM for PSI's infringement of the '709 patent, the '678 patent, the '520 patent, the '495 patent, and

the '993 patent pursuant to 35 U.S.C. § 284 and adequate to compensate IBM for PSI's breaches of the IBM license agreements pursuant to which PSI has licensed IBM software for purposes of its emulator development program;

e. That PSI be ordered to pay treble damages pursuant to 35 U.S.C. § 284;

f. That PSI be ordered to pay attorneys' fees pursuant to 35 U.S.C. § 285;

g. That PSI be ordered to pay prejudgment interest;

h. That PSI be ordered to pay all costs associated with this action; and

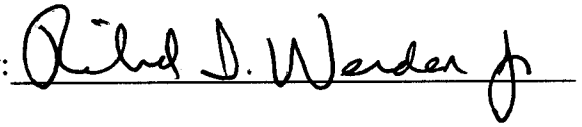
i. That IBM be granted such other and additional relief as the Court or a jury may deem just and proper.

**DEMAND FOR JURY TRIAL**

Pursuant to Rule 38 of the Federal Rules of Civil Procedure, IBM hereby demands a trial by jury as to all issues so triable.

DATED: New York, New York
November 29, 2006

QUINN EMANUEL URQUHART OLIVER & HEDGES, LLP

By: 

Richard I. Werder, Jr. (RW-5601)
Edward J. DeFranco (ED-6524)
David L. Elsberg (DE-9215)
51 Madison Avenue
22nd Floor
New York, New York 10010-1601
(212) 849-7000

Frederick A. Lorig
865 S. Figueroa Street
Los Angeles, California 90017
(213) 443-3000

Attorneys for Plaintiff
International Business Machines Corporation

# Customer Agreement

This IBM Customer Agreement (called the "Agreement") governs transactions by which you purchase Machines, license ICA Programs, obtain Program licenses, and acquire Services from International Business Machines Corporation ("IBM").

This Agreement and its applicable Attachments and Transaction Documents are the complete agreement regarding these transactions, and replace any prior oral or written communications between us.

By signing below for our respective Enterprises, both of us agree to the terms of this Agreement without modification. Once signed, 1) any reproduction of this Agreement, an Attachment, or Transaction Document made by reliable means (for example, photocopy or facsimile) is considered an original and 2) all Products and Services ordered under this Agreement are subject to it.

| | |
|---|---|
| *Agreed to:* | *Agreed to:* |
| **Customer Company name:** PLATFORM SOLUTIONS | **International Business Machines Corporation** |
| By _____ Authorized signature | By _____ Authorized signature |
| Name (type or print): Christopher Reilly | Name (type or print): Amadi A. Heyliger |
| Date: March 29th, 2004 | Date: 08-09-04 |
| Enterprise number: 7200934 | Agreement number: SU03963 |
| Enterprise address: 501 Macara Ave Suite 104 Sunnyvale CA 94085 | IBM address: 13800 Diplomat Drive Dallas, TX |

*After signing, please return a copy of this Agreement to the "IBM address" shown above.*

# IBM Customer Agreement

Table of Contents

# IBM Customer Agreement

## Part 1 - General

**1.1 Definitions**

**Customer-set-up Machine** is an IBM Machine that you install according to IBM's instructions.

**Date of Installation** is the following:

1. for an IBM Machine that IBM is responsible for installing, the business day after the day IBM installs it or, if you defer installation, makes it available to you for subsequent installation by IBM;
2. for a Customer-set-up Machine and a non-IBM Machine, the second business day after the Machine's standard transit allowance period; and
3. for a Program --
   a. basic license, the later of the following:
      (i) the day after its testing period ends; or
      (ii) the second business day after the Program's standard transit allowance period,
   b. copy, the date (specified in a Transaction Document) on which IBM authorizes you to make a copy of the Program, and
   c. chargeable component, the date you distribute a copy of the chargeable component in support of your authorized use of the Program.

**Designated Machine** is either 1) the machine on which you will use an ICA Program for processing and which IBM requires you to identify to it by type/model and serial number, or 2) any machine on which you use the ICA Program if IBM does not require you to provide this identification.

**Enterprise** is any legal entity (such as a corporation) and the subsidiaries it owns by more than 50 percent. The term "Enterprise" applies only to the portion of the Enterprise located in the United States.

**ICA Program** is an IBM Program licensed under Part 4 of this Agreement.

***Licensed Internal Code*** *(called "LIC") is Machine Code used by certain Machines IBM specifies (called "Specific Machines").*

**Machine** is a machine, its features, conversions, upgrades, elements, or accessories, or any combination of them. The term "Machine" includes an IBM Machine and any non-IBM Machine (including other equipment) that IBM may provide to you.

**Machine Code** is microcode, basic input/output system code (called "BIOS"), utility programs, device drivers, and diagnostics delivered with an IBM Machine.

**Materials** are literary works or other works of authorship (such as programs, program listings, programming tools, documentation, reports, drawings and similar works) that IBM may deliver to you as part of a Service. The term "Materials" does not include Programs, Machine Code, or LIC.

***Non-IBM Program is a Program licensed under a separate third party license agreement.***

**Other IBM Program** is an IBM Program licensed under a separate IBM license agreement, e.g., IBM International Program License Agreement.

**Product** is a Machine or a Program.

**Program** is the following, including the original and all whole or partial copies:

1. machine-readable instructions and data;
2. components;
3. audio-visual content (such as images, text, recordings, or pictures); and
4. related licensed materials.

The term "Program" includes any ICA Program, Other IBM Program, or Non-IBM Program that IBM may provide to you. *The term does not include Machine Code, LIC, or Materials.*

**Service** is performance of a task, provision of advice and counsel, assistance, support, or access to a resource (such as access to an information database) IBM makes available to you.

**Specifications** is a document that provides information specific to a Product. IBM provides an IBM Machine's Specifications in a document entitled "Official Published Specifications" and an ICA Program's Specifications in a document entitled "Licensed Program Specifications."

**Specified Operating Environment** is the Machines and programs with which an ICA Program is designed to operate, as described in the ICA Program's Specifications.

**1.2 Agreement Structure**

IBM provides additional terms for Products and Services in documents called "Attachments" and "Transaction Documents" which are also part of this Agreement. *All transactions have one or more associated* Transaction Documents (such as an invoice, supplement, schedule, exhibit, statement of work, change authorization, or addendum).

If there is a conflict among the terms in the various documents, those of an Attachment prevail over those of this Agreement. The terms of a Transaction Document prevail over those of both of these documents.

You accept the terms in Attachments and Transaction Documents by 1) signing them, 2) using the Product or Service, or allowing others to do so, or 3) making any payment for the Product or Service.

A Product or Service becomes subject to this Agreement when IBM accepts your order by 1) sending you a Transaction Document, 2) shipping the Machine or making the Program available to you, or 3) providing the Service.

**1.3 Delivery**

IBM will try to meet your delivery requirements for Products and Services you order, and will inform you of their status. Transportation charges, if applicable, will be specified in a Transaction Document.

**1.4 Charges and Payment**

The amount payable for a Product or Service will be based on one or more of the following types of charges: one-time, recurring, time and materials, or fixed price. Additional charges may apply (such as special handling or travel related expenses). IBM will inform you in advance whenever additional charges apply.

Recurring charges for a Product begin on its Date of Installation. Charges for Services are billed as IBM specifies which may be in advance, periodically during the performance of the Service, or after the Service is completed.

Services for which you prepay must be used within the applicable contract period. Unless IBM specifies otherwise, IBM does not give credits or refunds for unused prepaid Services.

**Charges**

One-time and recurring charges may be based on measurements of actual or authorized use (for example, number of users or processor size for Programs, meter readings for maintenance Services or connect time for network Services). You agree to provide actual usage data if IBM specifies. If you make changes to your environment that impact use charges (for example, change processor size or configuration for Programs), you agree to promptly notify IBM and pay any applicable charges. Recurring charges will be adjusted accordingly. Unless IBM agrees otherwise, IBM does not give credits or refunds for charges already due or paid. In the event that IBM changes the basis of measurement, its terms for changing charges will apply.

You receive the benefit of a decrease in charges for amounts which become due on or after the effective date of the decrease.

IBM may increase recurring charges for Products and Services, as well as labor rates and minimums for Services provided under this Agreement, by giving you three months' written notice. An increase applies on the first day of the invoice or charging period on or after the effective date IBM specifies in the notice.

IBM may increase one-time charges without notice. However, an increase to one-time charges does not apply to you if 1) IBM receives your order before the announcement date of the increase and 2) one of the following occurs within three months after IBM's receipt of your order:

1. IBM ships you the Machine or makes the Program available to you;
2. you make an authorized copy of a Program or distribute a chargeable component of a Program to another Machine; or
3. a Program's increased use charge becomes due.

**Payment**

Amounts are due upon receipt of invoice and payable as IBM specifies in a Transaction Document. You agree to pay accordingly, including any late payment fee.

If any authority imposes a duty, tax, levy, or fee, excluding those based on IBM's net income, upon any transaction under this Agreement, then you agree to pay that amount as specified in an invoice or supply exemption documentation. You are responsible for any personal property taxes for each Product from the date IBM ships it to you.

**1.5 Changes to the Agreement Terms**

In order to maintain flexibility in our business relationship, IBM may change the terms of this Agreement by giving you three months' written notice. However, these changes are not retroactive. They apply, as of the effective date IBM specifies in the notice, only to new orders, renewals, and on-going transactions that do not expire. For on-going transactions with a defined renewable contract period, you may request that IBM defer the change effective date until the end of the current contract period if 1) the change affects your current contract period and 2) you consider the change unfavorable. Changes to charges will be implemented as described in the Charges and Payment section above

Otherwise, for a change to be valid, both of us must sign it. Additional or different terms in any written communication from you (such as an order) are void.

**1.6 IBM Business Partners**

IBM has signed agreements with certain organizations (called "IBM Business Partners") to promote, market, and support certain Products and Services. When you order IBM Products or Services (marketed to you by IBM Business Partners) under this Agreement, IBM confirms that it is responsible for providing the Products or Services to you under the warranties and other terms of this Agreement. IBM is not responsible for 1) the actions of IBM Business Partners,

2) any additional obligations they have to you, or 3) any products or services that they supply to you under their agreements.

**1.7 Patents and Copyrights**

For purposes of this section, the term "Product" includes Materials, Machine Code and LIC.

If a third party claims that a Product IBM provides to you infringes that party's patent or copyright, IBM will defend you against that claim at its expense and pay all costs, damages, and attorney's fees that a court finally awards or that are included in a settlement approved by IBM, provided that you:

1. promptly notify IBM in writing of the claim; and
2. allow IBM to control, and cooperate with IBM in, the defense and any related settlement negotiations.

**Remedies**

If such a claim is made or appears likely to be made, you agree to permit IBM to enable you to continue to use the Product, or to modify it, or replace it with one that is at least functionally equivalent. If IBM determines that none of these alternatives is reasonably available, you agree to return the Product to IBM on its written request. IBM will then give you a credit equal to:

1. for a Machine, your net book value provided you have followed generally-accepted accounting principles;
2. for an ICA Program, the amount paid by you or 12 months' charges (whichever is less); and
3. for Materials, the amount you paid IBM for the creation of the Materials.

This is IBM's entire obligation to you regarding any claim of infringement.

**Claims for Which IBM is Not Responsible**

IBM has no obligation regarding any claim based on any of the following:

1. anything you provide which is incorporated into a Product or IBM's compliance with any designs, specifications, or instructions provided by you or by a third party on your behalf;
2. your modification of a Product, or an ICA Program's use in other than its Specified Operating Environment;
3. the combination, operation, or use of a Product with other products not provided by IBM as a system, or the combination, operation or use of a Product with any product, data, apparatus, or business method that IBM did not provide, or the distribution, operation or use of a Product for the benefit of a third party outside your Enterprise; or
4. infringement by a non-IBM Product or an Other IBM Program alone.

**1.8 Limitation of Liability**

Circumstances may arise where, because of a default on IBM's part or other liability, you are entitled to recover damages from IBM. In each such instance, regardless of the basis on which you are entitled to claim damages from IBM (including fundamental breach, negligence, misrepresentation, or other contract or tort claim), IBM is liable for no more than:

1. payments referred to in the Patents and Copyrights section above;
2. damages for bodily injury (including death) and damage to real property and tangible personal property; and
3. the amount of any other actual direct damages up to the greater of $100,000 or the charges (if recurring, 12 months' charges apply) for the Product or Service that is the subject of the claim. For purposes of this item, the term "Product" includes Materials, Machine Code, and LIC.

This limit also applies to any of IBM's subcontractors and Program developers. It is the maximum for which IBM and its subcontractors and Program developers are collectively responsible.

**Items for Which IBM is Not Liable**

Under no circumstances is IBM, its subcontractors, or Program developers liable for any of the following even if informed of their possibility:

1. loss of, or damage to, data;
2. special, incidental, or indirect damages or for any economic consequential damages; or
3. lost profits, business, revenue, goodwill, or anticipated savings.

**1.9 General Principles of Our Relationship**

1. Neither of us grants the other the right to use its (or any of its Enterprise's) trademarks, trade names, or other designations in any promotion or publication without prior written consent.
2. All information exchanged is nonconfidential. If either of us requires the exchange of confidential information, it will be made under a signed confidentiality agreement.
3. Each of us is free to enter into similar agreements with others.
4. Each of us grants the other only the licenses and rights specified. No other licenses or rights (including licenses or rights under patents) are granted.
5. Each of us may communicate with the other by electronic means and such communication is acceptable as a signed writing. An identification code (called a "user ID") contained in an electronic document is sufficient to verify the sender's identity and the document's authenticity.
6. Each of us will allow the other reasonable opportunity to comply before it claims that the other has not met its obligations.
7. Neither of us will bring a legal action arising out of or related to this Agreement more than two years after the cause of action arose.
8. Neither of us is responsible for failure to fulfill any obligations due to causes beyond its control.
9. Neither of us may assign this Agreement, in whole or in part, without the prior written consent of the other. Any attempt to do so is void. Neither of us will unreasonably withhold such consent. The assignment of this

Agreement, in whole or in part, within the Enterprise of which either of us is a part or to a successor organization by merger or acquisition does not require the consent of the other. IBM is also permitted to assign its rights to payments under this Agreement without obtaining your consent. It is not considered an assignment for IBM to divest a portion of its business in a manner that similarly affects all of its customers.

10. You agree not to resell any Service without IBM's prior written consent. Any attempt to do so is void.
11. You agree that this Agreement will not create any right or cause of action for any third party, nor will IBM be responsible for any third party claims against you except as described in the Patents and Copyrights section above or as permitted by the Limitation of Liability section above for bodily injury (including death) or damage to real or tangible personal property for which IBM is legally liable.
12. You agree to acquire Machines with the intent to use them within your Enterprise and not for reselling, leasing, or transferring to a third party, unless either of the following applies:
    a. you are arranging lease-back financing for the Machines; or
    b. you purchase them without any discount or allowance, and do not remarket them in competition with IBM's authorized remarketers.
13. You agree to allow IBM to install mandatory engineering changes (such as those required for safety) on a Machine. Any parts IBM removes become IBM's property. You represent that you have the permission from the owner and any lien holders to transfer ownership and possession of removed parts to IBM.
14. You agree that you are responsible for the results obtained from the use of the Products and Services.
15. You agree to provide IBM with sufficient, free, and safe access to your facilities and systems for IBM to fulfill its obligations.
16. You agree to allow International Business Machines Corporation and its subsidiaries to store and use your contact information, including names, phone numbers, and e-mail addresses, anywhere they do business. Such information will be processed and used in connection with our business relationship, and may be provided to contractors, Business Partners, and assignees of International Business Machines Corporation and its subsidiaries for uses consistent with their collective business activities, including communicating with you (for example, for processing orders, for promotions, and for market research).
17. You agree to comply with all applicable export and import laws and regulations.

**1.10 Agreement Termination**

Either of us may terminate this Agreement on written notice to the other following the expiration or termination of the terminating party's obligations.

Either of us may terminate this Agreement if the other does not comply with any of its terms, provided the one who is not complying is given written notice and reasonable time to comply.

Any terms of this Agreement which by their nature extend beyond the Agreement termination remain in effect until fulfilled, and apply to both of our respective successors and assignees.

**1.11 Geographic Scope and Governing Law**

The rights, duties, and obligations of each of us are valid only in the United States except that all licenses are valid as specifically granted.

Both you and IBM consent to the application of the laws of the State of New York to govern, interpret, and enforce all of your and IBM's rights, duties, and obligations arising from, or relating in any manner to, the subject matter of this Agreement, without regard to conflict of law principles.

In the event that any provision of this Agreement is held to be invalid or unenforceable, the remaining provisions of this Agreement remain in full force and effect.

Nothing in this Agreement affects any statutory rights of consumers that cannot be waived or limited by contract.

## Part 2 - Warranties

**2.1 The IBM Warranties**

**Warranty for IBM Machines**

IBM warrants that each IBM Machine is free from defects in materials and workmanship and conforms to its Specifications.

The warranty period for a Machine is a specified, fixed period commencing on its Date of Installation. During the warranty period, IBM provides repair and exchange Service for the Machine, without charge, under the type of Service IBM designates for the Machine. If a Machine does not function as warranted during the warranty period and IBM is unable to either 1) make it do so or 2) replace it with one that is at least functionally equivalent, you may return it to IBM and your money will be refunded.

Additional terms regarding Service for Machines during and after the warranty period are contained in Part 5.

**Warranty for ICA Programs**

IBM warrants that each warranted ICA Program, when used in the Specified Operating Environment, will conform to its Specifications.

The warranty period for an ICA Program expires when its Program Services are no longer available. During the warranty period, IBM provides defect-related Program Services without charge. Program Services are available for a warranted ICA Program for at least one year following its general availability.

If an ICA Program does not function as warranted during the first year after you obtain your license and IBM is unable to make it do so, you may return the ICA Program and your money will be refunded. To be eligible, you must have obtained your license while Program Services (regardless of the remaining duration) were available for it. Additional terms regarding Program Services are contained in Part 4.

**Warranty for IBM Services**
IBM warrants that it performs each IBM Service using reasonable care and skill and according to its current description (including any completion criteria) contained in this Agreement, an Attachment, or a Transaction Document.

**Warranty for Systems**
Where IBM provides Products to you as a system, IBM warrants that they are compatible and will operate with one another. This warranty is in addition to IBM's other applicable warranties.

**2.2 Extent of Warranty**
If a Machine is subject to federal or state consumer warranty laws, IBM's statement of limited warranty included with the Machine applies in place of these Machine warranties.

The warranties stated above will not apply to the extent that there has been misuse (including but not limited to use of any Machine capacity or capability, other than that authorized by IBM in writing), accident, modification, unsuitable physical or operating environment, operation in other than the Specified Operating Environment, improper maintenance by you, or failure caused by a product for which IBM is not responsible. With respect to Machines, the warranty is voided by removal or alteration of Machine or parts identification labels.

**THESE WARRANTIES ARE YOUR EXCLUSIVE WARRANTIES AND REPLACE ALL OTHER WARRANTIES OR CONDITIONS, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.**

**Items Not Covered by Warranty**
IBM does not warrant uninterrupted or error-free operation of a Product or Service or that IBM will correct all defects.

IBM will identify IBM Machines and ICA Programs that it does not warrant.

Unless IBM specifies otherwise, it provides Materials, non-IBM Products, and non-IBM Services **WITHOUT WARRANTIES OF ANY KIND**. However, non-IBM manufacturers, developers, suppliers, or publishers may provide their own warranties to you. Warranties, if any, for Other IBM Programs and Non-IBM Programs may be found in their license agreements.

## Part 3 - Machines

**3.1 Production Status**
Each IBM Machine is manufactured from parts that may be new or used. In some cases, a Machine may not be new and may have been previously installed. Regardless, IBM's appropriate warranty terms apply.

**3.2 Title and Risk of Loss**
When IBM accepts your order, IBM agrees to sell you the Machine described in a Transaction Document. IBM transfers title to you or, if you choose, your lessor when IBM ships the Machine. However, IBM reserves a purchase money security interest in the Machine until IBM receives the amounts due. For a feature, conversion, or upgrade involving the removal of parts which become IBM's property, IBM reserves a security interest until IBM receives payment of all the amounts due and the removed parts. You authorize IBM to file appropriate documents to permit IBM to perfect its purchase money security interest.

For each Machine, IBM bears the risk of loss or damage up to the time it is delivered to the IBM-designated carrier for shipment to you or your designated location. Thereafter, you assume the risk. Each Machine will be covered by insurance, arranged and paid for by IBM for you, covering the period until it is delivered to you or your designated location. For any loss or damage, you must 1) report the loss or damage in writing to IBM within 10 business days of delivery and 2) follow the applicable claim procedure.

**3.3 Installation**
You agree to provide an environment meeting the specified requirements for the Machine.

IBM has standard installation procedures. IBM will successfully complete these procedures before it considers an IBM Machine (other than a Machine for which you defer installation or a Customer-set-up Machine) installed.

You are responsible for installing a Customer-set-up Machine and, unless IBM agrees otherwise, a non-IBM Machine.

**Machine Features, Conversions and Upgrades**
IBM sells features, conversions and upgrades for installation on Machines, and, in certain instances, only for installation on a designated, serial-numbered Machine. Many of these transactions involve the removal of parts and their return to IBM. As applicable, you represent that you have the permission from the owner and any lien holders to 1) install features, conversions, and upgrades and 2) transfer ownership and possession of removed parts (which become IBM's

property) to IBM. You further represent that all removed parts are genuine, unaltered, and in good working order. A part that replaces a removed part will assume the warranty or maintenance Service status of the replaced part. You agree to allow IBM to install the feature, conversion, or upgrade within 30 days of its delivery. Otherwise, IBM may terminate the transaction and you must return the feature, conversion, or upgrade to IBM at your expense.

**3.4 Machine Code and LIC**

Machine Code is licensed under the terms of the agreement provided with the Machine Code. Machine Code is licensed only for use to enable a Machine to function in accordance with its Specifications and only for the capacity and capability for which you are authorized by IBM in writing and for which payment is received by IBM.

Certain Machines IBM specifies (called "Specific Machines") use LIC. IBM will identify Specific Machines in a Transaction Document. International Business Machines Corporation, one of its subsidiaries, or a third party owns LIC including all copyrights in LIC and all copies of LIC (this includes the original LIC, copies of the original LIC, and copies made from copies). LIC is copyrighted and licensed (not sold). LIC is licensed under the terms of the agreement provided with the LIC. LIC is licensed only for use to enable a Machine to function in accordance with its Specifications and only for the capacity and capability for which you are authorized by IBM in writing and for which payment is received by IBM.

## Part 4 - ICA Programs

**4.1 License**

When IBM accepts your order, IBM grants you a nonexclusive, nontransferable license to use the ICA Program in the United States. ICA Programs are owned by International Business Machines Corporation, one of its subsidiaries, or a third party and are copyrighted and licensed (not sold).

**Authorized Use**

Under each license, IBM authorizes you to:

1. use the ICA Program's machine-readable portion on only the Designated Machine. If the Designated Machine is inoperable, you may use another machine temporarily. If the Designated Machine cannot assemble or compile the ICA Program, you may assemble or compile the ICA Program on another machine.
   If you change a Designated Machine previously identified to IBM, you agree to notify IBM of the change and its effective date;
2. use the ICA Program to the extent of authorizations you have obtained;
3. make and install copies of the ICA Program, to support the level of use authorized, provided you reproduce the copyright notices and any other legends of ownership on each copy or partial copy, and
4. use any portion of the ICA Program IBM 1) provides in source form, or 2) marks restricted (for example, "Restricted Materials of IBM") only to –
   a. resolve problems related to the use of the ICA Program, and
   b. modify the ICA Program so that it will work together with other products.

**Your Additional Obligations**

For each ICA Program, you agree to:

1. comply with any additional terms in its Specifications or a Transaction Document;
2. ensure that anyone who uses it (accessed either locally or remotely) does so only for your authorized use and complies with IBM's terms regarding ICA Programs; and
3. maintain a record of all copies and provide it to IBM at its request.

**Actions You May Not Take**

You agree not to:

1. reverse assemble, reverse compile, or otherwise translate the ICA Program unless expressly permitted by applicable law without the possibility of contractual waiver; or
2. sublicense, assign, rent, or lease the ICA Program.

**4.2 Program Components Not Used on the Designated Machine**

Some ICA Programs have components that are designed for use on machines other than the Designated Machine on which the ICA Program is used. You may make copies of a component and its documentation in support of your authorized use of the ICA Program. For a chargeable component, you agree to notify IBM of its Date of Installation.

**4.3 Distributed System License Option**

For some ICA Programs, you may make a copy under a Distributed System License Option (called a "DSLO" copy). IBM charges less for a DSLO copy than for the original license (called the "Basic" license). In return for the lesser charge, you agree to do the following while licensed under a DSLO:

1. have a Basic license for the ICA Program;
2. provide problem documentation and receive Program Services (if any) only through the location of the Basic license; and
3. distribute to, and install on, the DSLO's Designated Machine, any release, correction, or bypass that IBM provides for the Basic license.

**4.4 Program Testing**

IBM provides a testing period for certain ICA Programs to help you evaluate if they meet your needs. If IBM offers a testing period, it will start 1) the second business day after the ICA Program's standard transit allowance period, or 2)

on another date specified in a Transaction Document. IBM will inform you of the duration of the ICA Program's testing period.

IBM does not provide testing periods for DSLO copies.

**4.5 Program Services**
IBM provides Program Services for warranted ICA Programs. If IBM can reproduce your reported problem in the Specified Operating Environment, IBM will issue defect correction information, a restriction, or a bypass. IBM provides Program Services for only the unmodified portion of a current release of an ICA Program.

IBM provides Program Services 1) on an on-going basis (with at least six months' written notice before IBM terminates Program Services), 2) until the date IBM specifies, or 3) for a period IBM specifies.

**4.6 License Termination**
You may terminate the license for an ICA Program on one month's written notice, or at any time during the ICA Program's testing period.

Licenses for certain replacement ICA Programs may be obtained for an upgrade charge. When you obtain licenses for these replacement ICA Programs, you agree to terminate the license of the replaced ICA Programs when charges become due, unless IBM specifies otherwise.

IBM may terminate your license if you fail to comply with the license terms. If IBM does so, your authorization to use the ICA Program is also terminated.

## Part 5 - Services

**5.1 Personnel**
Each of us is responsible for the supervision, direction, control, and compensation of our respective personnel.

IBM reserves the right to determine the assignment of its personnel.

IBM may subcontract a Service, or any part of it, to subcontractors selected by IBM.

**5.2 Materials Ownership and License**
IBM will specify Materials to be delivered to you. IBM will identify them as being "Type I Materials," "Type II Materials," or otherwise as we both agree. If not specified, Materials will be considered Type II Materials.

Type I Materials are those, created during the Service performance period, in which you will have all right, title, and interest (including ownership of copyright). IBM will retain one copy of the Materials. You grant IBM 1) an irrevocable, nonexclusive, worldwide, paid-up license to use, execute, reproduce, display, perform, distribute (internally and externally) copies of, and prepare derivative works based on, Type I Materials and 2) the right to authorize others to do any of the former.

Type II Materials are those, created during the Service performance period or otherwise (such as those that preexist the Service), in which IBM or third parties have all right, title, and interest (including ownership of copyright). IBM will deliver one copy of the specified Materials to you. IBM grants you an irrevocable, nonexclusive, worldwide, paid-up license to use, execute, reproduce, display, perform, and distribute, within your Enterprise only, copies of Type II Materials.

Each of us agrees to reproduce the copyright notice and any other legend of ownership on any copies made under the licenses granted in this section.

**5.3 Service for Machines (during and after warranty)**
IBM provides certain types of Service to keep Machines in, or restore them to, conformance with their Specifications. IBM will inform you of the available types of Service for a Machine. At its discretion, IBM will 1) either repair or exchange the failing Machine and 2) provide the Service either at your location or a service center.

When the type of Service requires that you deliver the failing Machine to IBM, you agree to ship it suitably packaged (prepaid unless IBM specifies otherwise) to a location IBM designates. After IBM has repaired or exchanged the Machine, IBM will return it to you at its expense unless IBM specifies otherwise. IBM is responsible for loss of, or damage to, your Machine while it is 1) in IBM's possession or 2) in transit in those cases where IBM is responsible for the transportation charges.

Any feature, conversion, or upgrade IBM services must be installed on a Machine which is 1) for certain Machines, the designated, serial-numbered Machine and 2) at an engineering-change level compatible with the feature, conversion, or upgrade.

IBM manages and installs selected engineering changes that apply to IBM Machines and may also perform preventive maintenance.

You agree to:
1. obtain authorization from the owner to have IBM service a Machine that you do not own; and
2. where applicable, before IBM provides Service --
   a. follow the problem determination, problem analysis, and service request procedures that IBM provides,
   b. secure all programs, data, and funds contained in a Machine, and
   c. inform IBM of changes in a Machine's location.

**Replacements**
When Service involves the exchange of a Machine or part, the item IBM replaces becomes its property and the replacement becomes yours. You represent that all removed items are genuine and unaltered. The replacement may not be new, but will be in good working order and at least functionally equivalent to the item replaced. The replacement assumes the warranty or maintenance Service status of the replaced item. Before IBM exchanges a Machine or part, you agree to remove all features, parts, options, alterations, and attachments not under IBM's service. You also agree to ensure that the item is free of any legal obligations or restrictions that prevent its exchange.

Some parts of IBM Machines are designated as Customer Replaceable Units (called, "CRUs"), e.g., keyboards, memory, or hard disk drives. IBM provides CRUs to you for replacement by you. You must return all defective CRUs to IBM within 30 days of your receipt of the replacement CRU. You are responsible for downloading designated Machine Code and LIC updates from an IBM Internet Web site or from other electronic media, and following the instructions that IBM provides.

**Items Not Covered**
Repair and exchange Services do not cover:
1. accessories, supply items, and certain parts, such as batteries, frames, and covers;
2. Machines damaged by misuse, accident, modification, unsuitable physical or operating environment, or improper maintenance by you;
3. Machines with removed or altered Machine or parts identification labels;
4. failures caused by a product for which IBM is not responsible;
5. service of Machine alterations; or
6. Service of a Machine on which you are using capacity or capability, other than that authorized by IBM in writing.

**Warranty Service Upgrade**
For certain Machines, you may select a Service upgrade from the standard type of warranty Service for the Machine. IBM charges for the Service upgrade during the warranty period.

You may not terminate the Service upgrade or transfer it to another Machine during the warranty period. When the warranty period ends, the Machine will convert to maintenance Service at the same type of Service you selected for warranty Service upgrade.

**5.4 Maintenance Coverage**
Whenever you order maintenance Service for Machines, IBM will inform you of the date on which maintenance Service will begin. IBM may inspect the Machine within one month following that date. If the Machine is not in an acceptable condition for service, you may have IBM restore it for a charge. Alternatively, you may withdraw your request for maintenance Service. However, you will be charged for any maintenance Service which IBM has performed at your request.

**5.5 Automatic Service Renewal**
Renewable Services renew automatically for a same length contract period unless either of us provides written notification (at least one month prior to the end of the current contract period) to the other of its decision not to renew.

**5.6 Termination and Withdrawal of a Service**
Either of us may terminate a Service if the other does not meet its obligations concerning the Service.

You may terminate a Service, on notice to IBM provided you have met all minimum requirements and paid any adjustment charges specified in the applicable Attachments and Transaction Documents. For a maintenance Service, you may terminate without adjustment charge provided any of the following circumstances occur:
1. you permanently remove the eligible Product, for which the Service is provided, from productive use within your Enterprise;
2. the eligible location, for which the Service is provided, is no longer controlled by you (for example, because of sale or closing of the facility); or
3. the Machine has been under maintenance Service for at least six months and you give IBM one month's written notice prior to terminating the maintenance Service.

You agree to pay IBM for 1) all Services IBM provides and any Products and Materials IBM delivers through Service termination, 2) all expenses IBM incurs through Service termination, and 3) any charges IBM incurs in terminating the Service.

IBM may withdraw a Service or support for an eligible Product on three months' written notice to you. If IBM withdraws a Service for which you have prepaid and IBM has not yet fully provided it to you, IBM will give you a prorated refund.

Any terms which by their nature extend beyond termination or withdrawal remain in effect until fulfilled and apply to respective successors and assignees.

US005696709A

# United States Patent [19]

Smith, Sr.

[11] Patent Number: 5,696,709

[45] Date of Patent: Dec. 9, 1997

[54] **PROGRAM CONTROLLED ROUNDING MODES**

[75] Inventor: **Ronald Morton Smith, Sr.**, Wappingers Falls, N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **414,866**

[22] Filed: **Mar. 31, 1995**

[51] **Int. Cl.**[6] ..... **G06F 7/38**

[52] **U.S. Cl.** ..... **364/745**; 364/748

[58] **Field of Search** ..... 364/745, 748, 364/715.04

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,823,260 4/1989 Imel et al. ..... 364/745 X

*Primary Examiner*—Tan V. Mai

*Attorney, Agent, or Firm*—Lynn L. Augspurger; David V. Rossi

[57] **ABSTRACT**

A computer system having a default floating point rounding mode that may be overridden by a rounding mode designated by an instruction. The current machine rounding mode is stored in a register, and an instruction includes a field for specifying whether rounding should be performed according to the current rounding mode or according to another rounding mode during execution thereof.

**12 Claims, 3 Drawing Sheets**

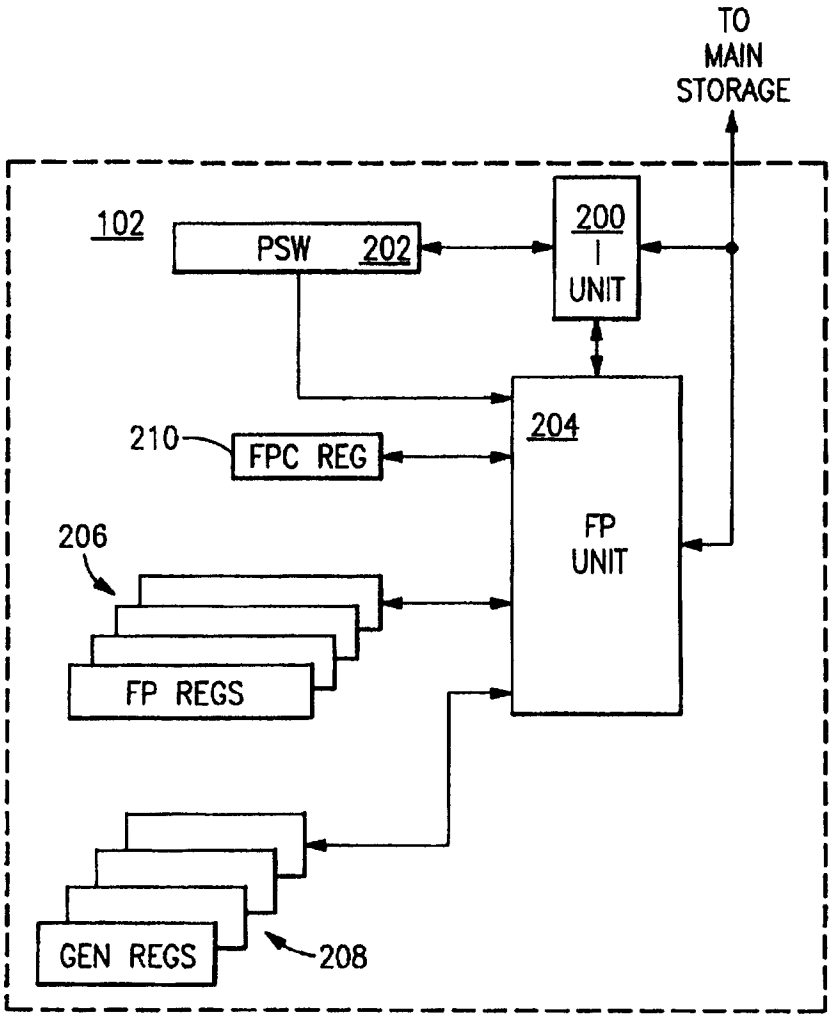


FIG.1

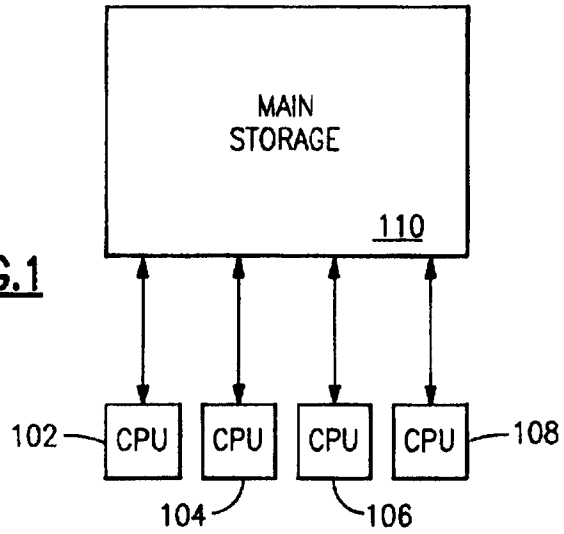
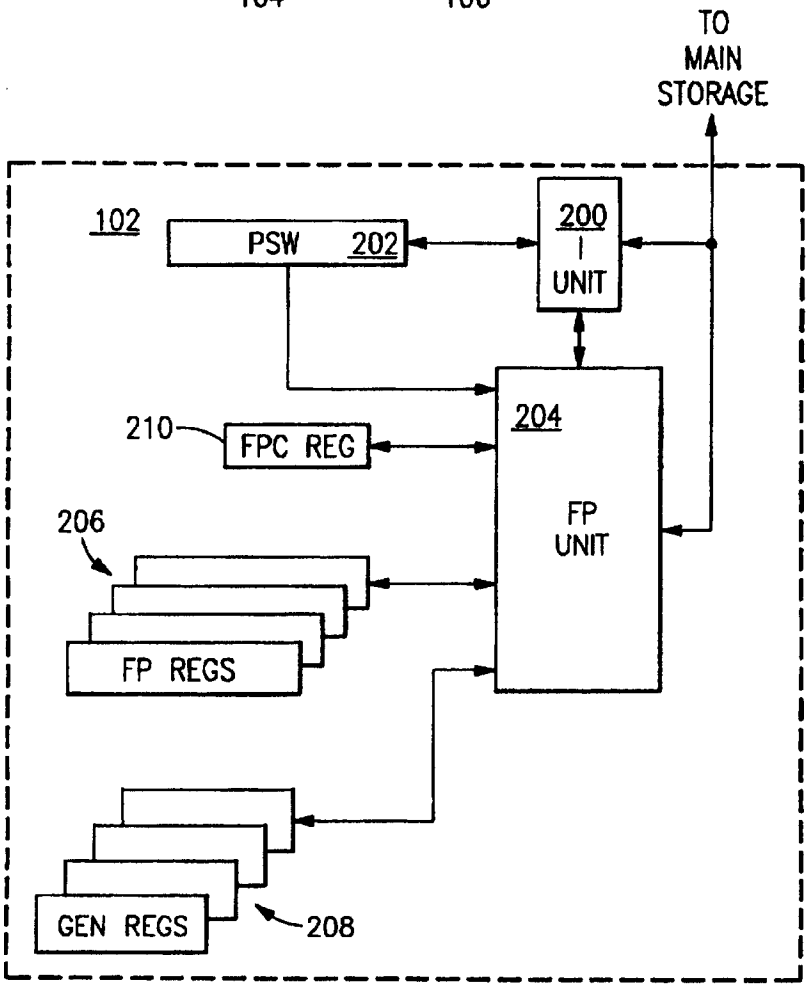

MAIN
STORAGE
110
102
CPU
CPU
CPU
CPU
108
104
106

FIG.2


TO
MAIN
STORAGE
102
PSW 202
200
I
UNIT
210
FPC REG
204
FP
UNIT
206
FP REGS
GEN REGS
208

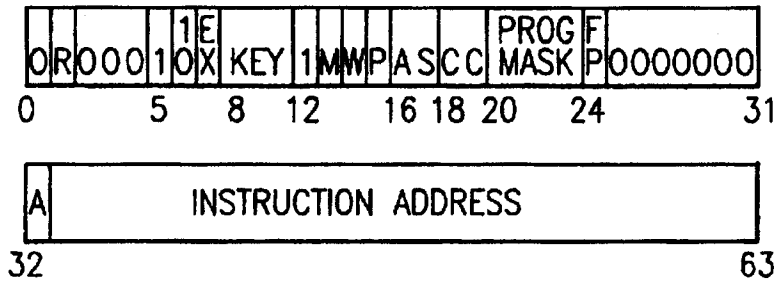
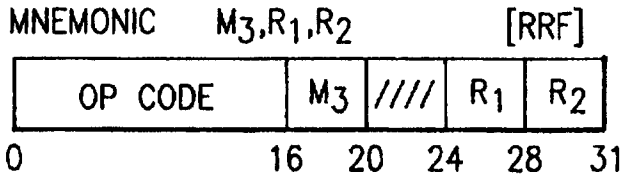
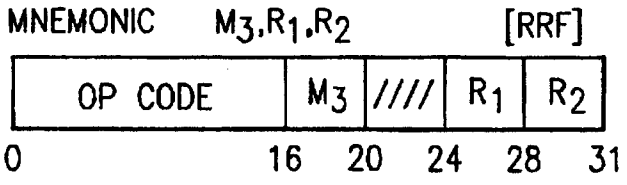



FIG.3




| MNEMONIC | OPERANDS |
|---|---|
| FIER | SHORT |
| FIDR | LONG |
| FIXR | EXTENDED |

FIG.5




| MNEMONIC | OPERANDS |
|---|---|
| CWER | SHORT OPERAND, 32–BIT BINARY–INTEGER RESULT |
| CWDR | LONG OPERAND, 32–BIT BINARY–INTEGER RESULT |

FIG.6

| BYTE | BITS | VALUE | FUNCTION |
|---|---|---|---|
| 0 | 0 | | BFP–INVALID–DATA MASK |
| | 1 | | BFP–DIVISION–BY–ZERO MASK |
| | 2 | | BFP–OVERFLOW MASK |
| | 3 | | BFP–UNDERFLOW MASK |
| | 4 | | BFP–INEXACT MASK |
| | 5–7 | | (UNASSIGNED) |
| 1 | 0 | | BFP–INVALID–DATA FLAG |
| | 1 | | BFP–DIVISION–BY–ZERO FLAG |
| | 2 | | BFP–OVERFLOW FLAG |
| | 3 | | BFP–UNDERFLOW FLAG |
| | 4 | | BFP–INEXACT FLAG |
| | 5–7 | | (UNASSIGNED) |
| 2 | 0–7 | | DATA–EXCEPTION CODE (DXC) |
| 3 | 0 | | QNaN MODE |
| | 1–5 | | (UNASSIGNED) |
| | 6–7 | | ROUNDING MODE |
| | | 00 | ROUND TO NEAREST |
| | | 01 | ROUND TO ZERO |
| | | 10 | ROUND UP |
| | | 11 | ROUND DOWN |

FIG.4

# PROGRAM CONTROLLED ROUNDING MODES

## FIELD OF THE INVENTION

The present invention relates to computer systems and, more particularly, to a computer architecture which includes instructions providing for programmable control of a rounding mode.

## BACKGROUND OF THE INVENTION

In the ensuing description of the prior art and the present invention, the following are herein incorporated by reference:

"Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-02, available through IBM branch offices, 1994;

"IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, August 1985; and

Commonly assigned U.S. patent application Ser. No. 08/414,250 to Eric Mark Schwarz, et al., filed Mar. 31, 1995, and entitled "Implementation of Binary Floating Point Using Hexadecimal Floating Point Unit".

In past architectures, rounding was provided either by means of a mode which controlled the rounding on all instructions, or by means of special rounding instructions. Each of these schemes has advantages and disadvantages. The mode has an advantage when a particular rounding mode is desired for an extended period of time. The special instructions have an advantage when a specific rounding is required for a single operation.

It would be advantageous, however, to have a machine which incorporates both a rounding mode and a rounding instruction.

## SUMMARY OF THE INVENTION

The present invention overcomes the above, and other, prior art limitations by providing a machine having a default rounding mode that may be overridden by a rounding mode designated by an instruction. The current machine rounding mode is stored in a register, and an instruction includes a field for specifying whether rounding should be performed according to the current rounding mode or according to another rounding mode during execution thereof.

## BRIEF DESCRIPTION OF THE DRAWINGS

Additional aspects, features, and advantages of the invention will be understood and will become more readily apparent when the invention is considered in the light of the following description made in conjunction with the accompanying drawings, wherein:

FIG. 1 illustrates a conventional shared memory computer system which may be employed to implement the present invention;

FIG. 2 schematically depicts functional components included in a CPU which may be employed in accordance with the present invention;

FIG. 3 illustrates the format of a 64 bit program status word (PSW), including a bit for indicating a binary or hexadecimal floating point mode, in accordance with an embodiment of the present invention;

FIG. 4 illustrates the format of a floating-point-control (FPC) register, including bits for indicating a rounding mode, in accordance with the present invention;



FIG. 5 illustrates the format of a LOAD FP Integer instruction, including a rounding mode field, in accordance with the present invention; and

FIG. 6 illustrates the format of a Convert to Fixed instruction, including a rounding mode field, in accordance with the present invention.

## DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 illustrates a conventional shared memory computer system including a plurality of central processing units (CPUs) **102–108** all having access to a common main storage **110**. FIG. 2 schematically depicts functional components included in a CPU from FIG. 1. Instruction unit **200** fetches instructions from common main storage **110** according to an instruction address located in the program status word (PSW) register **202**, and appropriately effects execution of these instructions. Instruction unit **200** appropriately hands off retrieved floating point instructions to floating point unit **204**, along with some of the operands that may be required by the floating point unit to execute the instruction. Floating point (FP) unit **204** includes all necessary hardware to execute the floating point instruction set, and preferably, in accordance with an embodiment of the present invention, supports both Binary and Hexadecimal floating point formats. FP unit **204** is coupled to floating point (FP) registers **206**, which contain floating point operands and results associated with FP unit **204** processing, and is also coupled to general registers **208**. FP unit **204** is also coupled to floating point control (FPC) register **210**, which preferably includes mask bits in addition to those provided in the PSW, as well as bits indicating the floating point mode. In a multi-user application, FPC register **210** is under control of the problem state.

FIG. 3 illustrates the format of a 64 bit PSW as stored in PSW register **202**. In a multi-user application, the supervisor state saves the PSW for a given problem state when taking interruption to dispatch another problem state. It can be seen that PSW includes program mask bits **20–23**.

FP-Mode Bit in PSW

Bit **24** of the PSW is the FP-mode bit. In accordance with an embodiment of the present invention whereby both binary and hexadecimal floating point modes are supported, when the bit is zero, the CPU is in the hexadecimal-floating-point (HFP) mode, and floating-point operands are interpreted according to the HFP format. When the bit is one, the CPU is the binary-floating-point (BFP) mode, and floating-point operands are assumed to be in the BFP format. Some floating-point instructions operate the same in either mode.

When an instruction is executed which is not available in the current FP mode, a special-operation exception is recognized.

FPC Register

As illustrated in detail by FIG. 4, the floating-point-control (FPC) register **210** is a 32-bit register, which contains the mode (i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last two bits of the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

Program Controlled Rounding Modes

In accordance with the present invention, the rounding mode indicated by the FPC register **210** may be superceded by certain instructions that are executed. Two instructions, LOAD FP INTEGER and CONVERT TO FIXED, are provided as examples of an embodiment of implementing program controlled rounding modes according to the present

invention, which is not limited thereto. FIG. 5 illustrates the format of a LOAD FP Integer instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand $R_2$ being rounded to an integer value in the same floating-point format, with the result placed in the first-operand location $R_1$ which identifies a floating point register 206. The resulting integer, which remains in floating-point format, either hexadecimal or binary, should not be confused with binary integers, which use a fixed-point format. If the floating-point operand is numeric with a large enough exponent so that it is already an integer, the

FPC Register

As illustrated in detail by FIG. 4, the floating-point-control (FPC) register 210 is a 32-bit register, which contains the mode (i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last to bits of the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

Program Controlled Rounding Modes

In accordance with the present invention, the rounding mode indicated by the FPC register 210 may be superceded by certain instructions that are executed. Two instructions, LOAD FP INTEGER and CONVERT TO FIXED, are provided as examples of an embodiment of implementing program controlled rounding modes according to the present invention, which is not limited thereto. FIG. 5 illustrates the format of a LOAD FP Integer instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand $R_2$ being rounded to an integer value in the same floating-point format, with the result placed in the first-operand location $R_1$ which identifies a floating point register 206. The resulting integer, which remains in floating-point format, either hexadecimal or binary, should not be confused with binary integers, which use a fixed-point format. If the floating-point operand is numeric with a large enough exponent so that it is already an integer, the result value remains the same, except that, in the HFP mode, an unnormalized operand is normalized, and an operand with a zero fraction is changed to a true zero.

In accordance with an embodiment of the present invention, a modifier in the $M_3$ field controls the method of rounding in the BFP mode. The second operand, if numeric, is rounded to an integer value as specified by the modifier in the $M_3$ field:

$M_3$ Rounding Method

0 According to current rounding mode

1 Biased round to nearest

4 Round to nearest

5 Round to zero

6 Round up

7 Round down

When the modifier field is zero, rounding is controlled by the current rounding mode in the FPC register. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4–7 is the same as for rounding modes 0–3 (binary 00–11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude. It may be understood that, in accordance with an embodiment of the present invention where both hexadecimal and binary floating point are supported, if the modifier is 5, the mehtod of rounding is the same in the HFP and BFP modes.

FIG. 6 illustrates the format of a Convert to Fixed instruction which may be executed by FP Unit 204. Execution of this instruction results in a floating point number located in a FP register 206 identified by the second operand $R_2$ being converted to a binary-integer, fixed-point format, with the result placed in the first-operand location $R_1$ which identifies a general register 208.

The result of CWDR and CWER is a 32-bit signed binary integer that is placed in the general register designated by $R_1$. A modifier in the $M_3$ field controls the method of rounding.

If the second operand is numeric, finite, and not already an integer, it is converted to an integer value in the fixed-point format by rounding as specified by the modifier in the $M_3$ field:

$M_3$ Rounding Method

0 According to current rounding mode

1 Biased round to nearest

4 Round to nearest

5 Round to zero

6 Round up

7 Round down

When the modifier field is zero, rounding is controlled by the current rounding mode in the FPC register 210. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4–7 is the same as for rounding modes 0–3 (binary 00–11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude.

A modifier other than 0, 1, or 4–7 is invalid. The sign of the result is the sign of the second operand, except that a zero result has a plus sign. Note that if the modifier is 5, the method of rounding is the same in the HFP and BFP modes.

Although the above description provides many specificities, these enabling details should not be construed as limiting the scope of the invention, and it will be readily understood by those persons skilled in the an that the present invention is susceptible to many modifications, adaptations, and equivalent implementations without departing from this scope and without diminishing its attendant advantages. It is therefore intended that the present invention is not limited to the disclosed embodiments but should be defined in accordance with the claims which follow.

What is claimed is:

1. A computer system, comprising:

a storage device including at least one stored bit for specifying one of plurality of rounding modes as a default rounding mode; and

a processor that executes a floating point instruction having a field that is operative to selectively override said default rounding mode with another of said rounding modes during execution of said floating point instruction by said processor;

wherein said rounding modes each specify a respective rounding direction applied to a floating point number.

2. The computer system according to claim 1, wherein said processor supports binary floating point format and hexadecimal floating point format.

3. The computer system according to claim 2, wherein said floating point instruction is common to said binary floating point format and said hexadecimal floating point format.

4. The computer system according to claim 1, wherein said floating point rounding modes include rounding modes specified by IEEE Std 754-1985 standards.

5. The computer system according to claim 1, wherein said floating point rounding modes include a biased round to nearest mode.

6. The computer system according to claim 1, wherein said floating point instruction includes a convert to fixed instruction.

7. The computer system according to claim 1, wherein said floating point instruction includes a load floating point integer instruction.

8. The computer system according to claim 1, wherein said field has a value which indicates that said default rounding mode is operative during execution of said floating point instruction by said processor.

9. A computer system including a plurality of floating point rounding modes, each of said floating point rounding modes specifying a respective rounding direction applied to a floating point number, said system comprising:

a storage element containing a value that specifies one of a plurality of rounding modes as a default rounding mode;

an instruction including a modifier field that selectively indicates another of said plurality of rounding modes; and

a processor that executes said instruction to provide a result of a given accuracy which is generated by rounding a floating point number having greater accuracy than the result, the rounding executed according to said default rounding mode when said modifier field does not override said default mode, and the rounding executed according to said another rounding mode when said modifier field overrides said default rounding mode, the rounding thereby being executed in response to said modifier field of said instruction.

10. The computer system according to claim 9, wherein said modifier field includes a value which indicates that said default rounding mode is operative during execution of said instruction by said processor.

11. A computer system including a plurality of floating point rounding modes, each of said floating point rounding modes specifying a respective rounding direction applied to a floating point number, said system comprising:

means for storing a value for specifying one of said floating point rounding modes as a default rounding mode; and

means for executing a floating point instruction having a field that is operative to selectively override said default rounding mode with another of said rounding modes during execution of said floating point instruction by said processor.

12. The computer system according to claim 11, wherein said field has a value which indicates that said default rounding mode is operative during execution of said floating point instruction.

* * * * *

US005825678A

# United States Patent [19]

Smith

[11] Patent Number: 5,825,678

[45] Date of Patent: Oct. 20, 1998

[54] **METHOD AND APPARATUS FOR DETERMINING FLOATING POINT DATA CLASS**

[75] Inventor: **Ronald M. Smith**, Wrappingers Falls, N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **414,858**

[22] Filed: **Mar. 31, 1995**

[51] **Int. Cl.**[6] ..... **G06F 7/38**; G06F 7/00; G06F 15/00; H03M 7/00

[52] **U.S. Cl.** ..... **364/748**; 364/715.03; 341/50; 341/94

[58] **Field of Search** ..... 364/748, 715.03; 341/50, 89, 104, 105, 94; 395/375, 800

[56] **References Cited**

U.S. PATENT DOCUMENTS

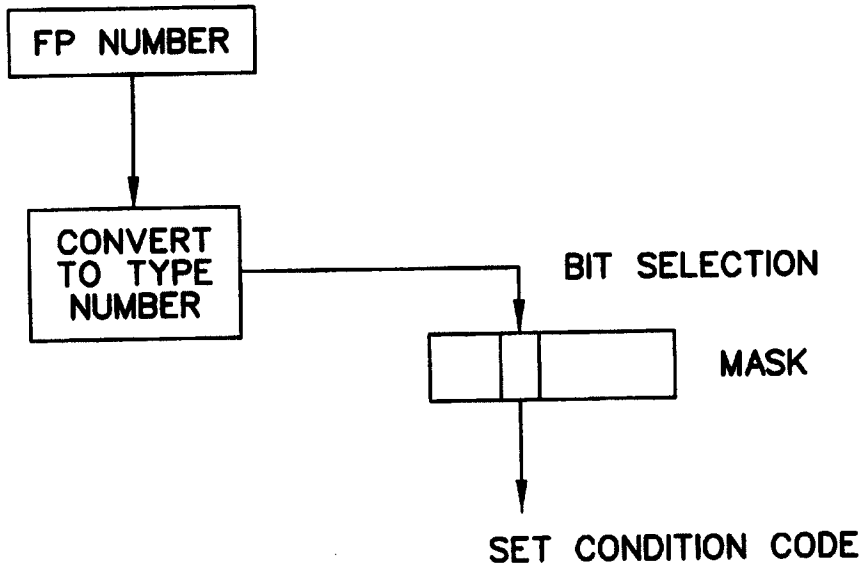
| | | | |
|---|---|---|---|
| 4,325,120 | 4/1982 | Colley et al. | 395/412 |
| 4,707,783 | 11/1987 | Lee et al. | 395/375 |
| 4,831,575 | 5/1989 | Kuroda | 364/748 |
| 5,191,335 | 3/1993 | Leitherer | 341/94 |

*Primary Examiner*—Reba I. Elmore
*Assistant Examiner*—Emmanuel L. Moise
*Attorney, Agent, or Firm*—Lynn Augspurger, Esq.; Morgan & Finnegan, LLP

[57] **ABSTRACT**

A new Test FP Data Class operation is provided which utilizes a 12-bit mask to determine to which of the 12 possible data classes a floating point number belongs and sets a condition code accordingly. As preferably embodied, a typical IBM System 390 instruction format is adapted to implement a Test FP Data Class operation. The class and sign of the first operand are examined to select one bit from the second-operand address. A condition code of 0 or 1 is set according to whether the selected bit is 0 or 1. The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand calls and sign.

**12 Claims, 6 Drawing Sheets**



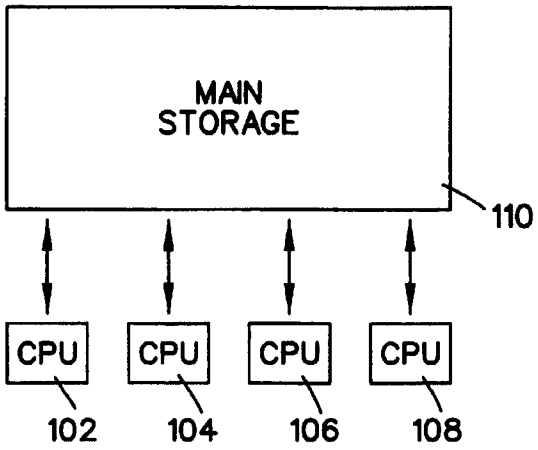

MAIN
STORAGE
110
CPU
CPU
CPU
CPU
102
104
106
108

FIG. 1

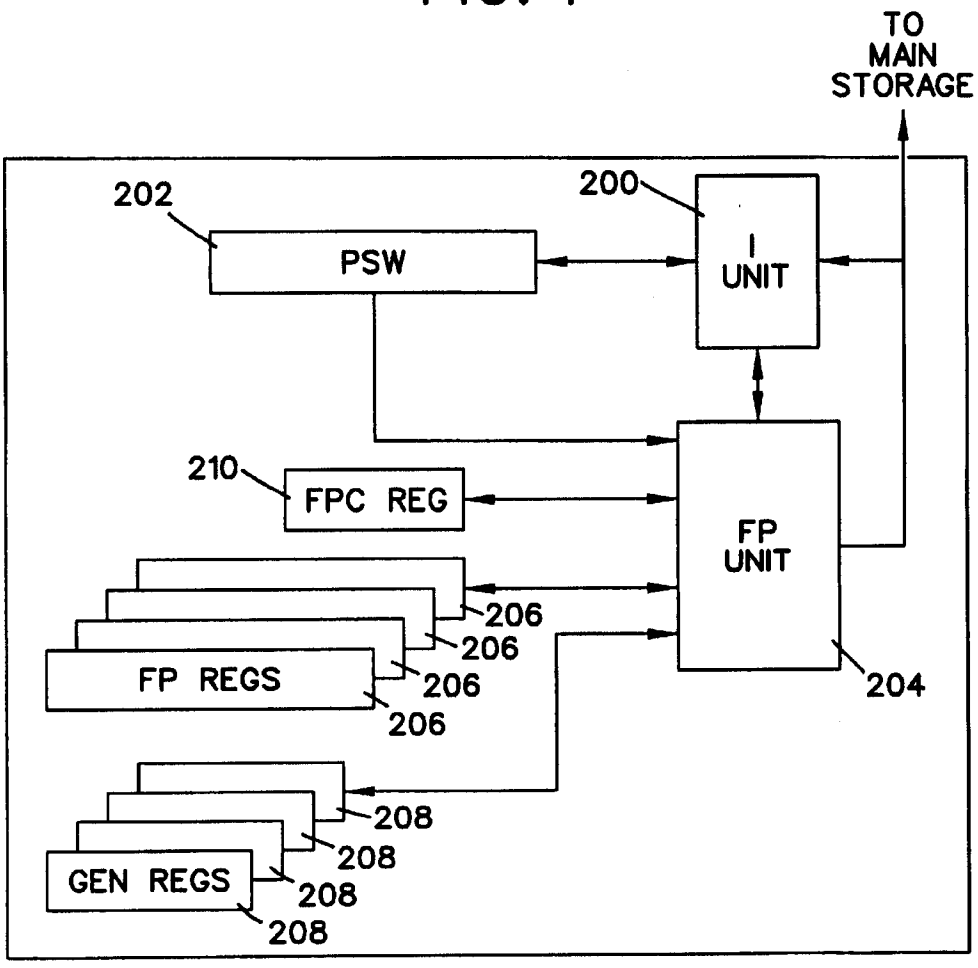

TO
MAIN
STORAGE
202
200
PSW
I
UNIT
210
FPC REG
FP
UNIT
206
206
206
206
FP REGS
204
208
208
208
208
GEN REGS

FIG. 2

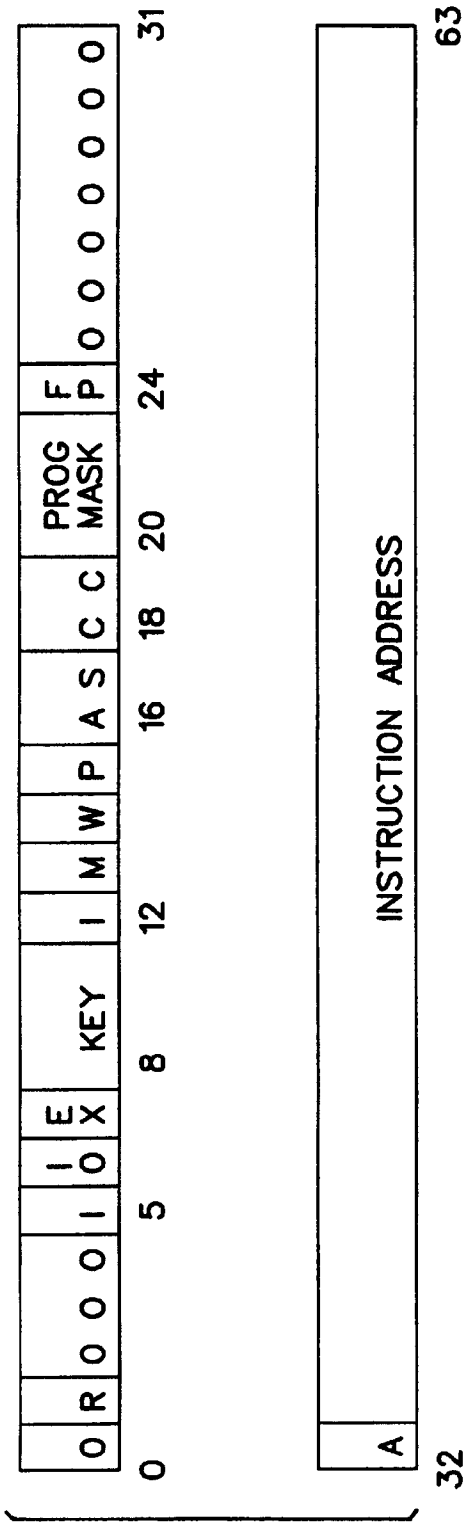

0
5
8
12
16
18
20
24
31
63
32
O
R
I
I/O
EX
KEY
I
M
W
P
A S
C C
PROG MASK
F P
INSTRUCTION ADDRESS
A

FIG. 3

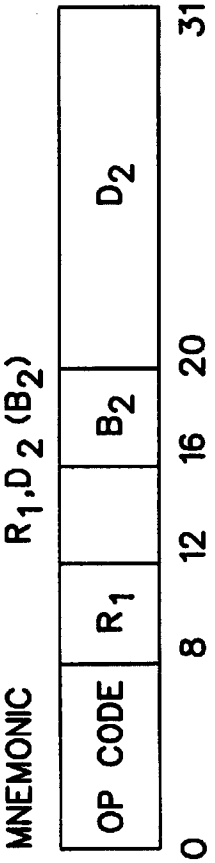

MNEMONIC
R1,D2 (B2)
0
8
12
16
20
31
OP CODE
R1
B2
D2

FIG. 5

| BYTE | BITS | VALUE | FUNCTION |
|---|---|---|---|
| 0 | 0 | | BFP-INVALID-DATA MASK |
| | 1 | | BFP-DIVISION-BY-ZERO MASK |
| | 2 | | BFP-OVERFLOW MASK |
| | 3 | | BFP-UNDERFLOW MASK |
| | 4 | | BFP-INEXACT MASK |
| | 5-7 | | (UNASSIGNED) |
| 1 | 0 | | BFP-INVALID-DATA FLAG |
| | 1 | | BFP-DIVISION-BY-ZERO FLAG |
| | 2 | | BFP-OVERFLOW FLAG |
| | 3 | | BFP-UNDERFLOW FLAG |
| | 4 | | BFP-INEXACT FLAG |
| | 5-7 | | (UNASSIGNED) |
| 2 | 0-7 | | DATA-EXCEPTION CODE (DXC) |
| 3 | 0 | | QNaN MODE |
| | 1-5 | | (UNASSIGNED) |
| | 6-7 | | ROUNDING MODE |
| | | 00 | ROUND TO NEAREST |
| | | 01 | ROUND TO ZERO |
| | | 10 | ROUND UP |
| | | 11 | ROUND DOWN |

FIG.4

| ENTITY | SIGN | BIASED EXPONENT | UNIT BIT * | FRACTION |
|---|---|---|---|---|
| TRUE ZERO | ± | 0 | 0 | 0 |
| DENORMALIZED NUMBERS | ± | 0 ** | 0 | NOT 0 |
| NORMALIZED NUMBERS | ± | NOT 0, NOT ALL ONES | 1 | ANY |
| INFINITY | ± | ALL ONES | - | 0 |
| QUIET NaN | ± | ALL ONES | - | FO=1, Fr=ANY |
| SIGNALING NaN | ± | ALL ONES | - | FO=0, Fr≠0 |

EXPLANATION:

| | |
|---|---|
| * | THE UNIT BIT IS IMPLIED |
| ** | THE BIASED EXPONENT IS TREATED ARITHMETICALLY AS IF IT HAD THE VALUE ONE |
| NaN | NOT A NUMBER |
| FO | LEFTMOST BIT OF FRACTION |
| Fr | REMAINING BITS OF FRACTION |

FIG.6

| BFP OPERAND CLASS | BIT USED WHEN SIGN IS | |
|---|---|---|
| | + | − |
| ZERO | 20 | 21 |
| NORMALIZED NUMBER | 22 | 23 |
| DENORMALIZED NUMBER | 24 | 25 |
| INFINITY | 26 | 27 |
| QUIET NaN | 28 | 29 |
| SIGNALING NaN | 30 | 31 |

FIG.7

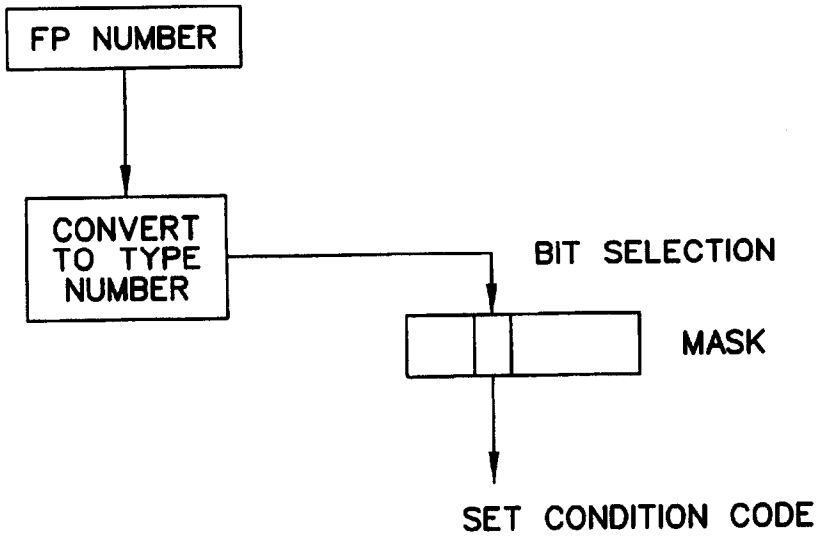



FIG.8

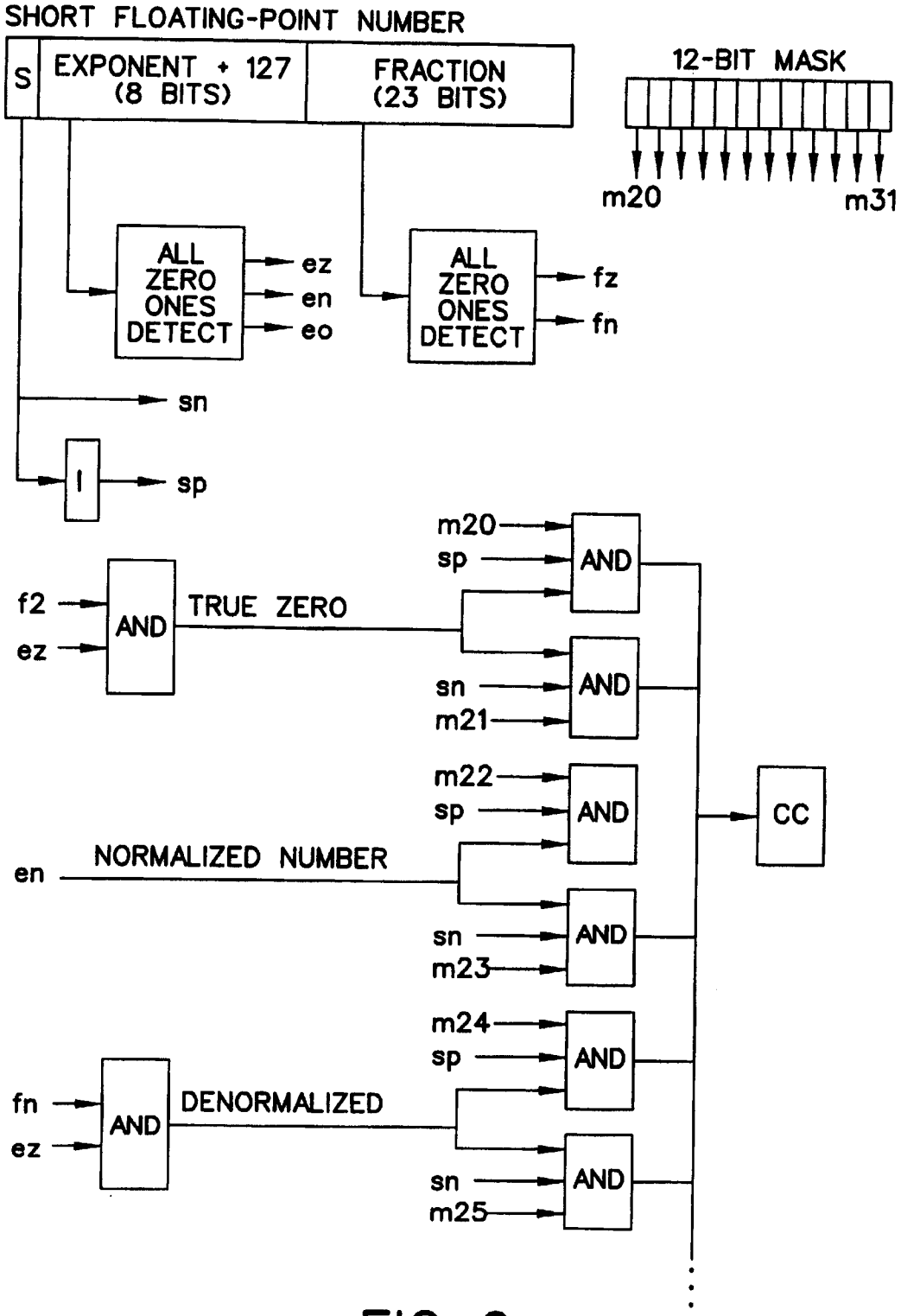

SHORT FLOATING-POINT NUMBER
S
EXPONENT + 127 (8 BITS)
FRACTION (23 BITS)
12-BIT MASK
m20
m31
ALL ZERO ONES DETECT
ez
en
eo
ALL ZERO ONES DETECT
fz
fn
sn
1
sp
m20
sp
AND
f2
ez
AND
TRUE ZERO
sn
m21
AND
m22
sp
AND
CC
en
NORMALIZED NUMBER
sn
m23
AND
m24
sp
AND
fn
ez
AND
DENORMALIZED
sn
m25
AND

FIG. 9

# METHOD AND APPARATUS FOR DETERMINING FLOATING POINT DATA CLASS

## BACKGROUND OF THE INVENTION

The present invention is directed to computer architecture. More particularly, the present invention is directed to computer architecture implementing floating point operations.

The present application generally relies upon the following as background in describing the invention and the prior art:

"Enterprise Systems Architecture/390 Principles of operation" (1994), Order No. SA22-7201-02, available from International Business Machines Corporation of Armonk, N.Y.;

"IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754–1985" (August 1985), available from The Institute of Electrical and Electronic Engineers, Inc., New York, New York; and

U.S. patent application Ser. No. 08/414,866 entitled "Implementation of Binary Floating Point Using Hexadecimal Floating Point Unit" filed on Mar. 31, 1995, in the name of Eric Mark Schwarz, et al., and assigned to International Business Machines Corporation of Armonk, N.Y.

The descriptions in the foregoing references are incorporated herein in their entirety by reference.

Although previous hardware implementations of floating-point arithmetic have provided various radixes, including binary, decimal, or hexadecimal, only a single radix was supported in any particular implementation. As future machines are built, however, they must be compatible with previous machines and must also provide support for new formats. Thus, a new requirement emerges to provide hardware support for more than one format. In particular, there is a requirement to support both the IBM System/360 hexadecimal and the IEEE binary floating-point formats. This results in several unique problems which must be solved.

Current instructions, such as Load And Test, which test the state of a floating point number, set the condition code to indicate the sign and value of the number. With IEEE floating-point numbers, there are 12 possible combinations of value and sign. This number of combinations, however, cannot be accommodated in the condition code of the program status word (PSW).

Accordingly, there is a need in the art for an operation that will provide more complete information on the data class of a floating point number.

## SUMMARY OF THE INVENTION

With the foregoing in mind, it is an object of the invention to provide a new Test FP Data Class operation which utilizes a 12-bit mask to determine to which of the 12 possible data classes a floating point number belongs and sets the condition code accordingly.

As preferably embodied, a typical IBM System 390 instruction format is adapted to implement a Test FP Data Class operation. In the Test FP Data Class instruction, $R_1$ corresponds to the floating point register to be tested, and $B_2$ to the base and $D_2$ to the offset used to form the second operand address. The class and sign of the first operand ($R_1$) are examined to select one bit from the second-operand address ($D_2(B_2)$). A condition code of 0 or 1 is set according to whether the selected bit is 0 or 1.



The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand calls and sign. In BFP mode, the rightmost 12 bits of the address, bits **20–31**, are used to specify 12 combinations of operand class and sign; bits **0–19** of the second-operand class are ignored.

## BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, referred to herein and constituting a part hereof, illustrate preferred embodiments of the invention and, together with the description, serve to explain the principles of the invention, wherein:

FIG. **1** illustrates a conventional shared memory computer system;

FIG. **2** illustrates functional components included in a CPU from FIG. **1**;

FIG. **3** illustrates the format of a 64 bit program status word;

FIG. **4** illustrates a floating-point-control register;

FIG. **5** illustrates a typical instruction format as is used, e.g., in an IBM System 390 computing system;

FIG. **6** shows the definition of each of the six classes of BFP data;

FIG. **7** illustrates the second operand address bits used for the Test FP Data Class operation in BFP mode;

FIG. **8** illustrates the execution of the Test FP Data Class operation; and

FIG. **9** illustrates the conversion circuitry of the convert to type number logic of the Test FP Data Class operation.

## DETAILED DESCRIPTION OF THE INVENTION

FIG. **1** illustrates a conventional shared memory computer system including a plurality of central processing units (CPUs) **102–108** all having access to a common main storage **110**.

FIG. **2** illustrates functional components included in a CPU from FIG. **1**. Instruction unit **200** fetches instructions from common main storage **110** according to an instruction address located in the program status word (PSW) register **202**, and appropriately effects execution of these instructions. Instruction unit **200** appropriately hands off retrieved floating point instructions to floating point unit **204**, along with some of the operands that may be required by the floating point unit to execute the instruction. Floating point (FP) unit **204** includes all necessary hardware to execute the floating point instruction set, and preferably, in accordance with an embodiment of the present invention, supports both Binary and Hexadecimal floating point formats. FP unit **204** is coupled to floating point (FP) registers **206**, which contain floating point operands and results associated with FP unit **204** processing, and is also coupled to general registers **208**. FP unit **204** is also coupled to floating point control (FPC) register **210**, which preferably includes mask bits in addition to those provided in the PSW. In a multi-user application, FPC register **210** is under control of the problem state program.

FIG. **3** illustrates the format of a 64 bit PSW as stored in PSW register **202**. In a multi-user application, the supervisor state program saves the PSW for a given problem state program when taking interruption to dispatch another problem state program. It can be seen that PSW includes program mask bits **20–23**

Bit **24** of the PSW is the FP-mode bit. In accordance with an embodiment of the present invention whereby both

binary and hexadecimal floating point modes are supported, when the bit is zero, the CPU is in the hexadecimal-floating-point (HFP) mode, and floating-point operands are interpreted according to the HFP format. When the bit is one, the CPU is in the binary-floating-point (BFP) mode, and floating-point operands are assumed to be in BFP format. Some floating-point instructions operate the same in either mode.

When an instruction is executed which is not available in the current FP mode, a special-operation exception is recognized.

As illustrated in FIG. **4**, the floating-point-control (FPC) register **210** is a 32-bit register, which contains the mode ( i.e., rounding mode), mask, flag, and code bits. For this implementation, by way of example, the rounding mode is represented by the last two bits o f the last byte. Round to nearest, round to zero, round up, and round down modes are supported.

In FIG. **5** is illustrated a typical instruction format as is used, e.g., in an IBM System 390 computing system. As preferably embodied, this instruction format is adapted to implement the Test FP Data Class operation described herein. Further, three particular types of instructions for the Test FP Data Class operation may be implemented, i.e., mneumonics TCE, TCD, TCX corresponding to short, long, and extended operands, respectively. As adapted to the Test FP Data Class operation herein, $R_1$ designates the floating point register to be tested, and $B_2$ designates the base and $D_2$ is the offset used to form the second operand address. The class and sign of the first operand ($R_1$,) are examined to select one bit from the second-operand address ($D_2(B_2)$). Condition code 0 or 1 is set according to whether the selected bit is 0 or 1.

The second-operand address is not used to address data; instead, individual bits of the address are used to specify the applicable combinations of operand class and sign. In BFP mode, the rightmost 12 bits of the address, bits **20–31**, are used to specify 12 combinations of operand class and sign; bits **0–19** of the second-operand class are ignored. Thus, as preferably embodied, $B_2$ may point to a general register which contains mask bits and $D_2$ would contain all zeros. Alternatively, when the $B_2$ field is all zeros, no base register is selected and $D_2$ may contain the mask bits. A third alternative where neither have all zeros is also possible, but not as practical.

FIG. **6** shows the definition of each of the six classes of BFP data including true zero, denormalized numbers, normalized numbers, infinity, quiet NaN, and signaling NaN. FIG. **7** illustrates the second operand address bits used for the Test FP Data Class operation in BFP mode.

In operation, a floating point processor conforming to IEEE standards provides an indication of the class of floating point number, which could be any one of the BFP data classes shown in FIG. **6**. It may be appreciated that a program may want to know whether the floating point number is characterized by some combination of these BFP data classes. The Test FP Data Class operation provides the ability with one instruction to check all of the BFP data classes that the floating point number may fall into.

As illustrated in FIG. **8**, a 12-bit mask is set, a bit corresponding to each of the possible classes and a particular bit is set to check for a particular data class. The floating point number in the floating point register that is pointed to by $R_1$ is loaded into the convert to type number logic and a determination is made as to the data class of the number. In the convert to type number logic, these floating point numbers are classified according to the sign, the fraction part, and the exponent part of the floating point number. That is, the floating point format has sign, exponent, and fraction bit. Based on these three bits, the floating point number may be categorized. Based upon the particular data class that it falls into, a signal is generated that will indicate one of the bits in the mask. That bit is then loaded into the condition code. That is, the condition code is set.

FIG. **9** illustrates the conversion circuitry of the convert to type number logic for bits **20** through **25** of the mask for the Test FP Data Class operation testing the short format (TCE). It will be appreciated that similar circuitry is used for bits **26** through **31** and for long (TCD) and extended (TCX) formats. In FIG. **9**, the following abbreviations are used:

cc—condition code

en—exponent not zero and not all ones

eo—exponent all ones

ez—exponent zero

fn—fraction not zero

fz—fraction zero

sn—sign negative

sp—sign plus

By way of example, if it was to be determined whether the number is positive zero, bit location **20** in the mask field would be set to 1. Then, if the floating point number were a positive zero the convert to type number logic would cause the mask bit **1** to be loaded into the condition code storage location. If a positive zero is not to be tested, a 0 would be set in bit location **20** in the mask field and the 0 would be loaded into the condition code storage location.

In view of the foregoing, it may be appreciated that by simply checking one bit in the PSW, namely the condition code, any combination of the twelve data classes that the floating point number might fall into may be determined.

It may be further appreciated that more of the second-operand-address bits may be set to one. If the second-operand-address bit corresponding to the class and sign of the first operand is one, condition code 1 is set; otherwise, condition code 0 is set.

It may be further appreciated that operands, including signaling NaNs and quiet NaNs, may be examined without causing an arithmetic exception.

It may be further appreciated that the Test FP Data Class operation provides a way to test an operand without risk of an exception or setting BFP flags.

It may be further appreciated that the second-operand-address bits assigned for zero and normalized numbers are the same in the HFP and BFP modes. Thus, programs which use the Test FP Data Class operation to test only for these operand classes may be written in a mode-independent manner.

While the invention has been described in its preferred embodiments, it is to be understood that the words which have been used are words of description, rather than limitation, and that changes may be made within the purview of the appended claims without departing from the true scope and spirit of the invention in its broader aspects.

What is claimed:

1. An apparatus for determining floating point data class, comprising:

a floating point processor for interpreting a machine instruction to determine whether the data class of a floating point number is an identified data class;

means for retrieving the floating point number from memory;

means for determining whether the data class of the floating point number is the identified data class by examination of condition of the fields of the floating point number; and

means for setting a condition code in a program status word based upon the determination of whether the data class is the identified data class.

2. An apparatus for determining floating point data class in accordance with claim 1, wherein said means for determining uses a bit mask to determine action to be taken for a particular data class of the floating point number.

3. An apparatus for determining floating point data class in accordance with claim 2, wherein said machine instruction includes the location of said floating point number in memory and said bit mask.

4. An apparatus for determining floating point data class in accordance with claim 1, wherein said means for determining operates in a mode independent manner.

5. An apparatus for determining floating point data class in accordance with claim 1, wherein said machine instruction can accommodate floating point numbers having a short, long, or extended operand.

6. An apparatus according to claim 1, wherein said identified data class includes any specified combination of a plurality of possible data classes.

7. A method for determining floating point data class, comprising the steps of:

interpreting a machine instruction to determine whether the data class of a floating point number is an identified data class;

retrieving the floating point number from memory;

determining whether the data class of the floating point number is the identified data class by examination of condition of the fields of the floating point number; and

setting a condition code in a program status word based upon the determination of whether the data class is the identified data class.

8. A method for determining floating point data class in accordance with claim 7, wherein said step of determining further comprises the step of using a bit mask to determine action to be taken for a particular data class of the floating point number.

9. A method for determining floating point data class in accordance with claim 8, further comprising the step of utilizing a machine instruction including location of the floating point number in memory and the bit mask.

10. A method for determining floating point data class in accordance with claim 7, wherein said step of determining is accomplished in a mode independent manner.

11. A method for determining floating point data class in accordance with claim 7, further comprising the step of using a machine instruction accommodating floating point numbers having a short, long, or extended operand.

12. A method according to claim 7, wherein said identified data class includes any specified combination of a plurality of possible data classes.

* * * * *

US005953520A

# United States Patent [19]

Mallick

[11] Patent Number: 5,953,520

[45] Date of Patent: Sep. 14, 1999

[54] **ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE**

[75] Inventor: **Soummya Mallick**, Austin, Tex.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **08/934,645**

[22] Filed: **Sep. 22, 1997**

[51] **Int. Cl.**[6] ..... **G06F 9/455**

[52] **U.S. Cl.** ..... **395/500.47**

[58] **Field of Search** ..... 395/568, 385, 395/500

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,339,417 | 8/1994 | Connell et al. | 395/650 |
| 5,574,873 | 11/1996 | Davidian | 395/376 |
| 5,742,802 | 4/1998 | Harter et al. | 395/568 |
| 5,790,825 | 11/1995 | Traut | 395/385 |

*Primary Examiner*—Kevin J. Teska

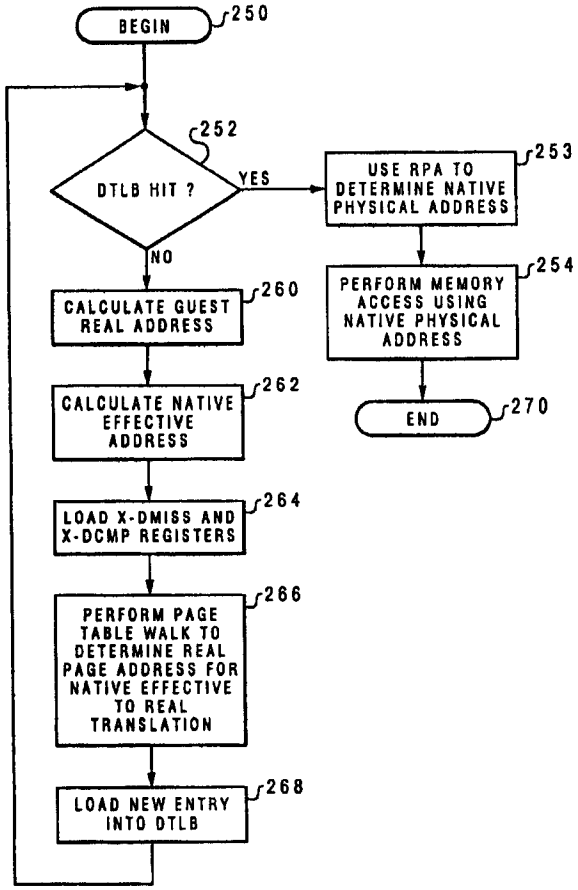
*Assistant Examiner*—Ayni Mohamed

*Attorney, Agent, or Firm*—Casimer K. Salys; Brian F. Russell; Andrew J. Dillon

[57] **ABSTRACT**

A processor and method of operating a processor which has a native instruction set and emulates instructions in a guest instruction set are described. According to the method, a series of guest instructions from the guest instruction set are stored in memory. The series includes a guest memory access instruction that indicates a guest logical address in guest address space. For each guest instruction in the series, a semantic routine of native instructions from the native instruction set is stored in memory. The semantic routines, which utilize native addresses in native address space, can be executed in order to emulate the guest instructions. In response to receipt of the guest memory access instruction for emulation, the guest logical address is translated into a guest real address, which is thereafter translated into a native physical address. A semantic routine that emulates the guest memory access instruction is then executed utilizing the native physical address.

**16 Claims, 9 Drawing Sheets**



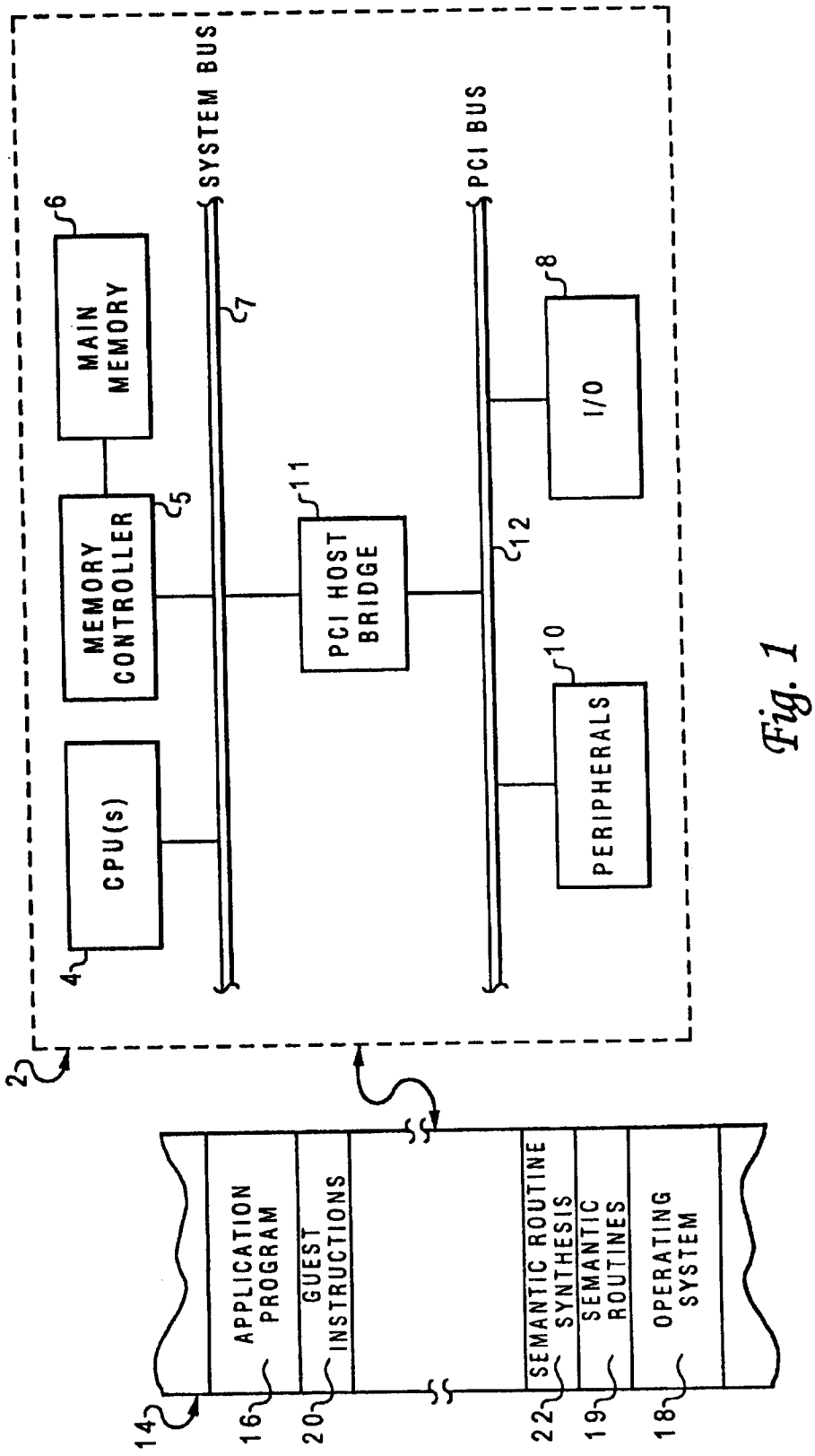

SYSTEM BUS
PCI BUS
MAIN MEMORY
MEMORY CONTROLLER
CPU(s)
PCI HOST BRIDGE
I/O
PERIPHERALS
6
5
4
7
11
12
8
10
2
APPLICATION PROGRAM
GUEST INSTRUCTIONS
SEMANTIC ROUTINE SYNTHESIS
SEMANTIC ROUTINES
OPERATING SYSTEM
14
16
20
22
19
18

Fig. 1

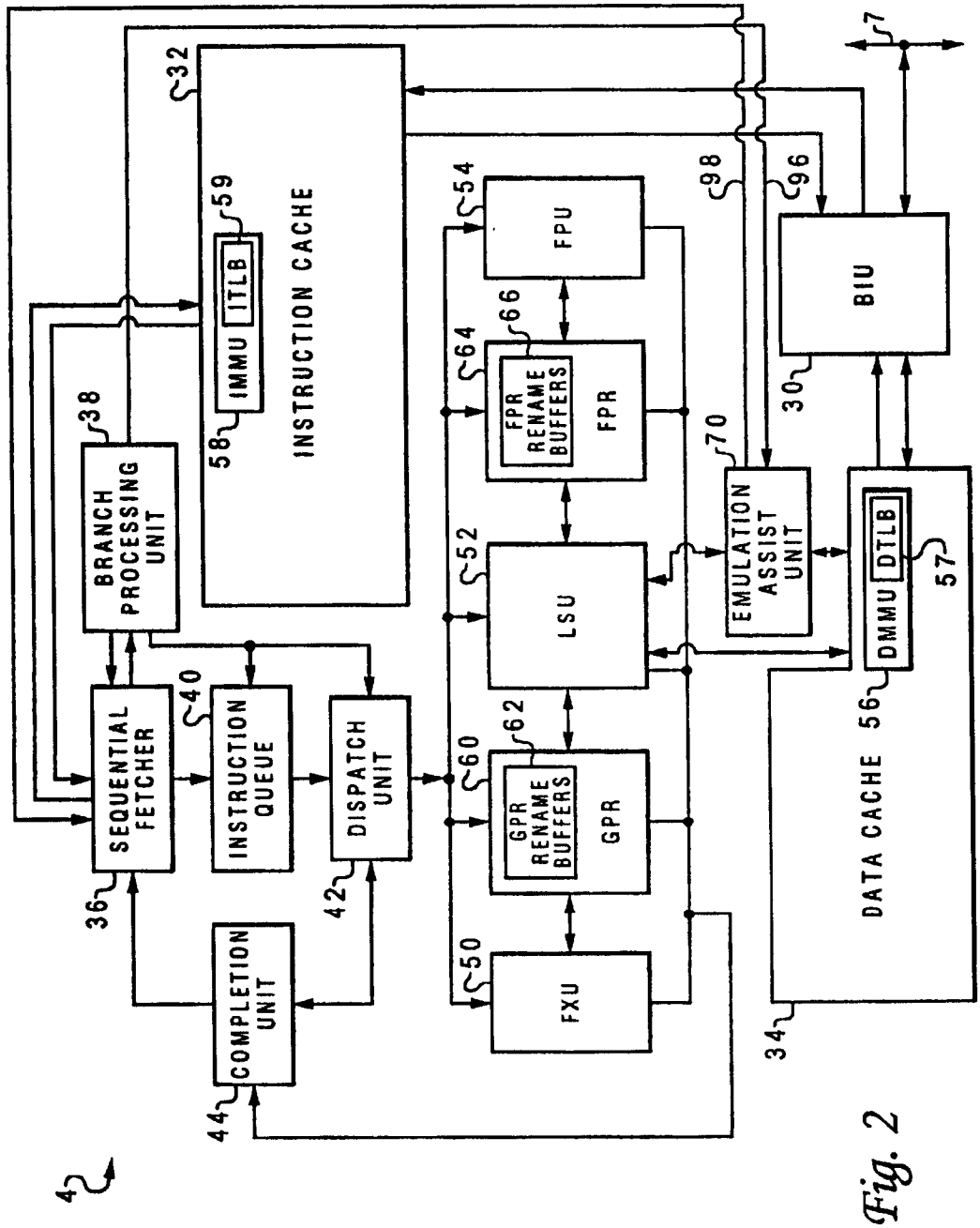

42
32
INSTRUCTION CACHE
59
ITLB
IMMU
58
38
BRANCH PROCESSING UNIT
SEQUENTIAL FETCHER
36
40
INSTRUCTION QUEUE
DISPATCH UNIT
42
COMPLETION UNIT
44
54
FPU
66
64
FPR RENAME BUFFERS
FPR
52
LSU
62
60
GPR RENAME BUFFERS
GPR
50
FXU
98
96
BIU
30
70
EMULATION ASSIST UNIT
DMMU
DTLB
57
DATA CACHE
56
34
7

Fig. 2

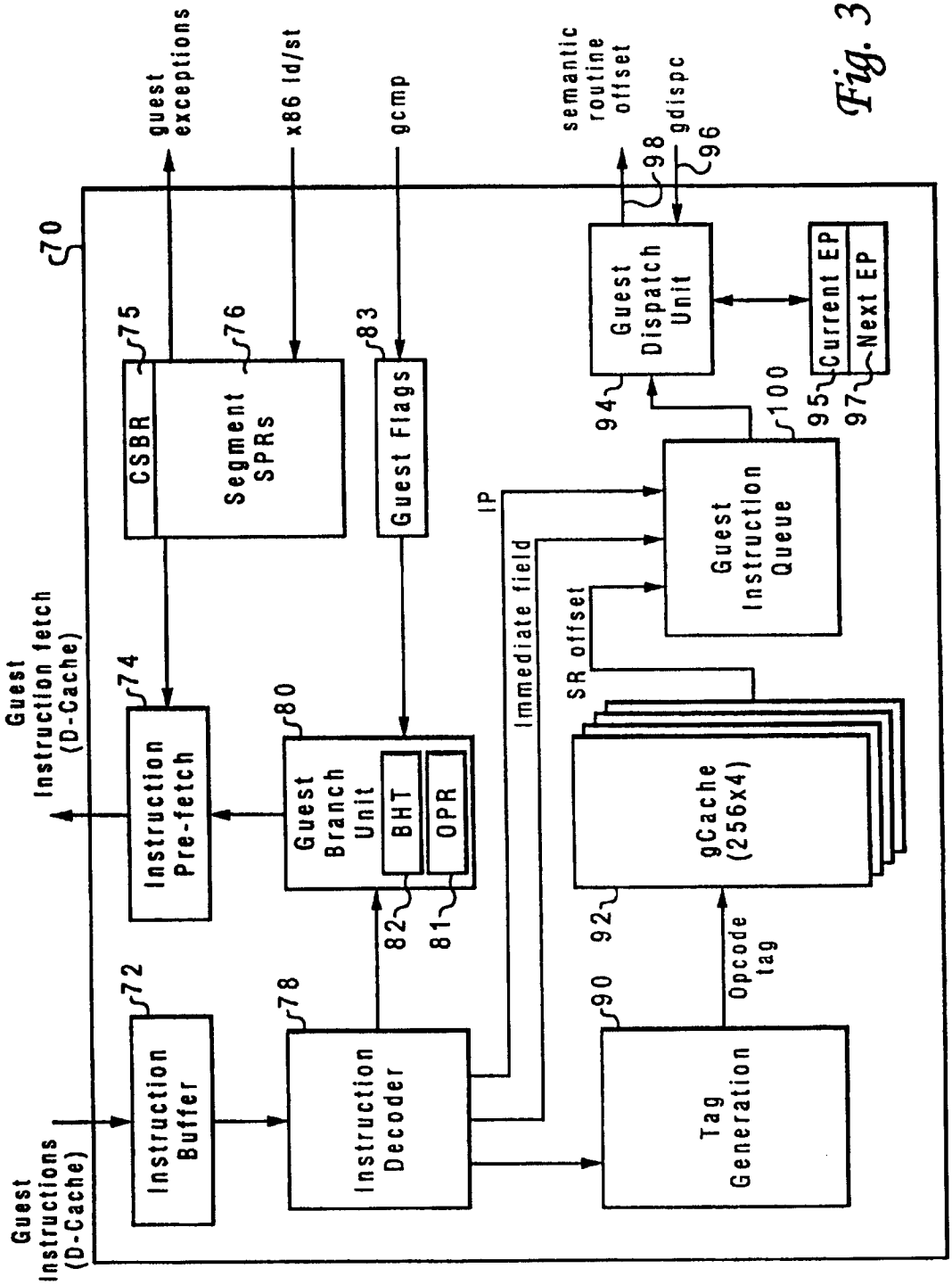

guest exceptions
x86 ld/st
gcmp
semantic routine offset
gdispc
98
96
70
75
76
83
CSBR
Segment SPRs
Guest Flags
Guest Dispatch Unit
Current EP
Next EP
94
100
95
97
IP
Immediate field
Guest Instruction Queue
SR offset
Guest Instruction fetch (D-Cache)
74
80
Instruction Pre-fetch
Guest Branch Unit
BHT
OPR
gCache (256x4)
82
81
92
72
78
90
Opcode tag
Instruction Buffer
Instruction Decoder
Tag Generation
Guest Instructions (D-Cache)

Fig. 3

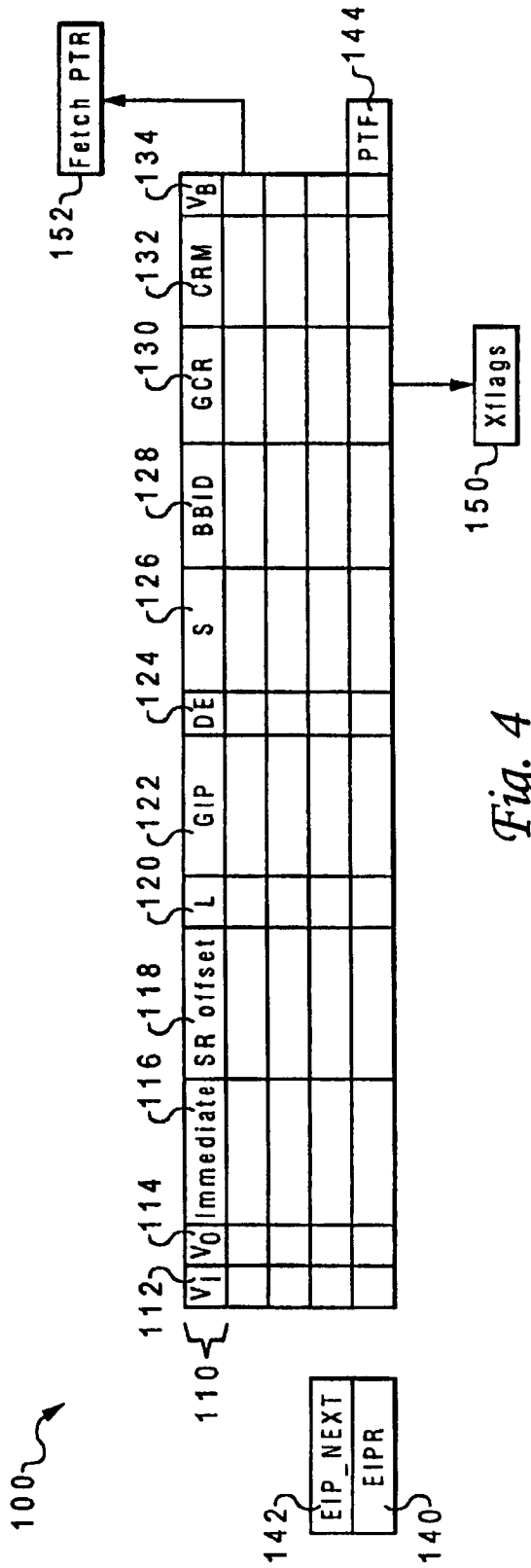

Fetch PTR
152
144
PTF
134
$V_B$
132
CRM
130
GCR
128
BBID
126
S
124
DE
122
GIP
120
L
118
SR offset
116
Immediate
114
$V_0$
112
$V_1$
110
100
Xflags
150
EIP_NEXT
EIPR
142
140

*Fig. 4*

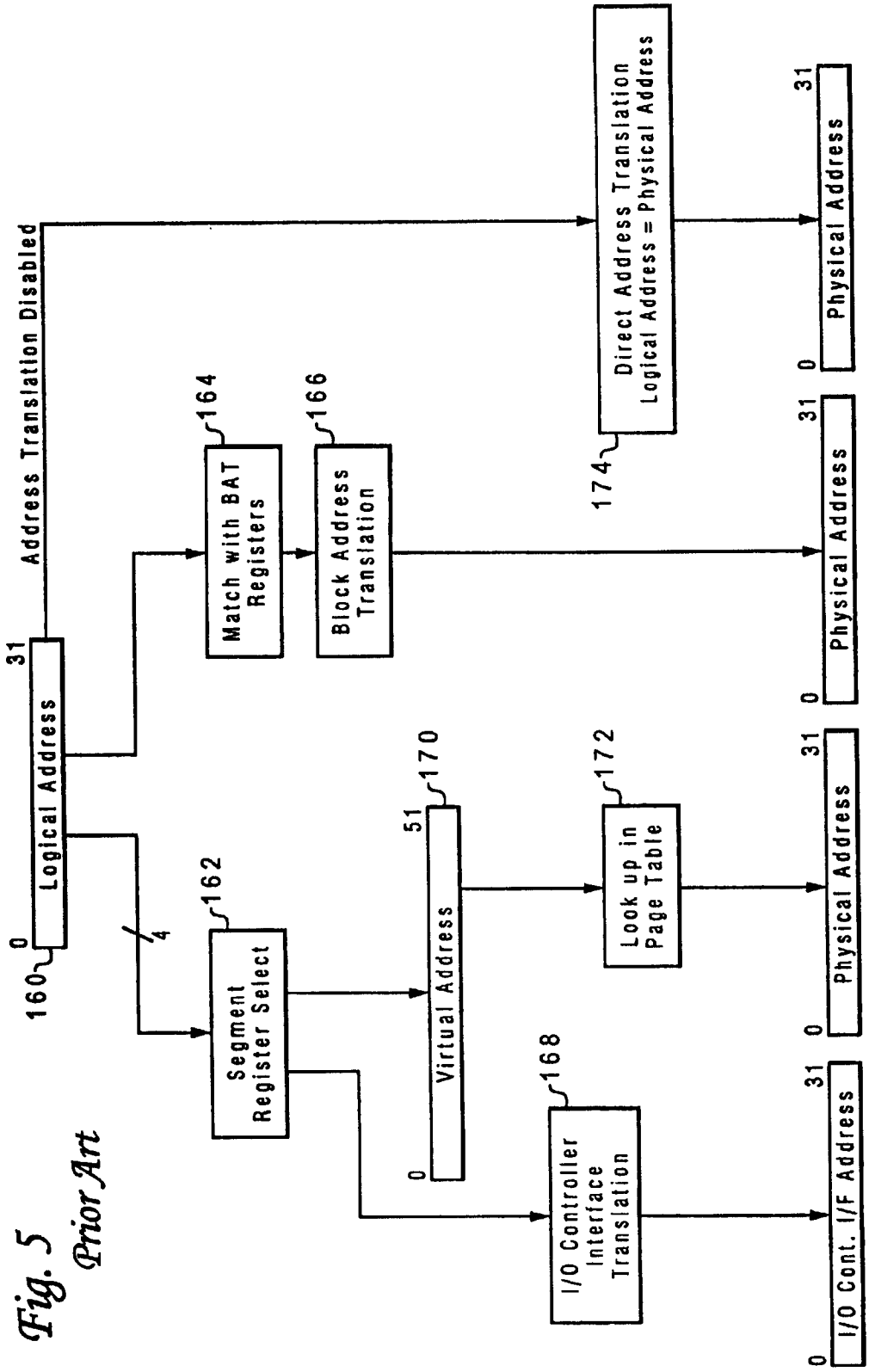



Fig. 5
Prior Art

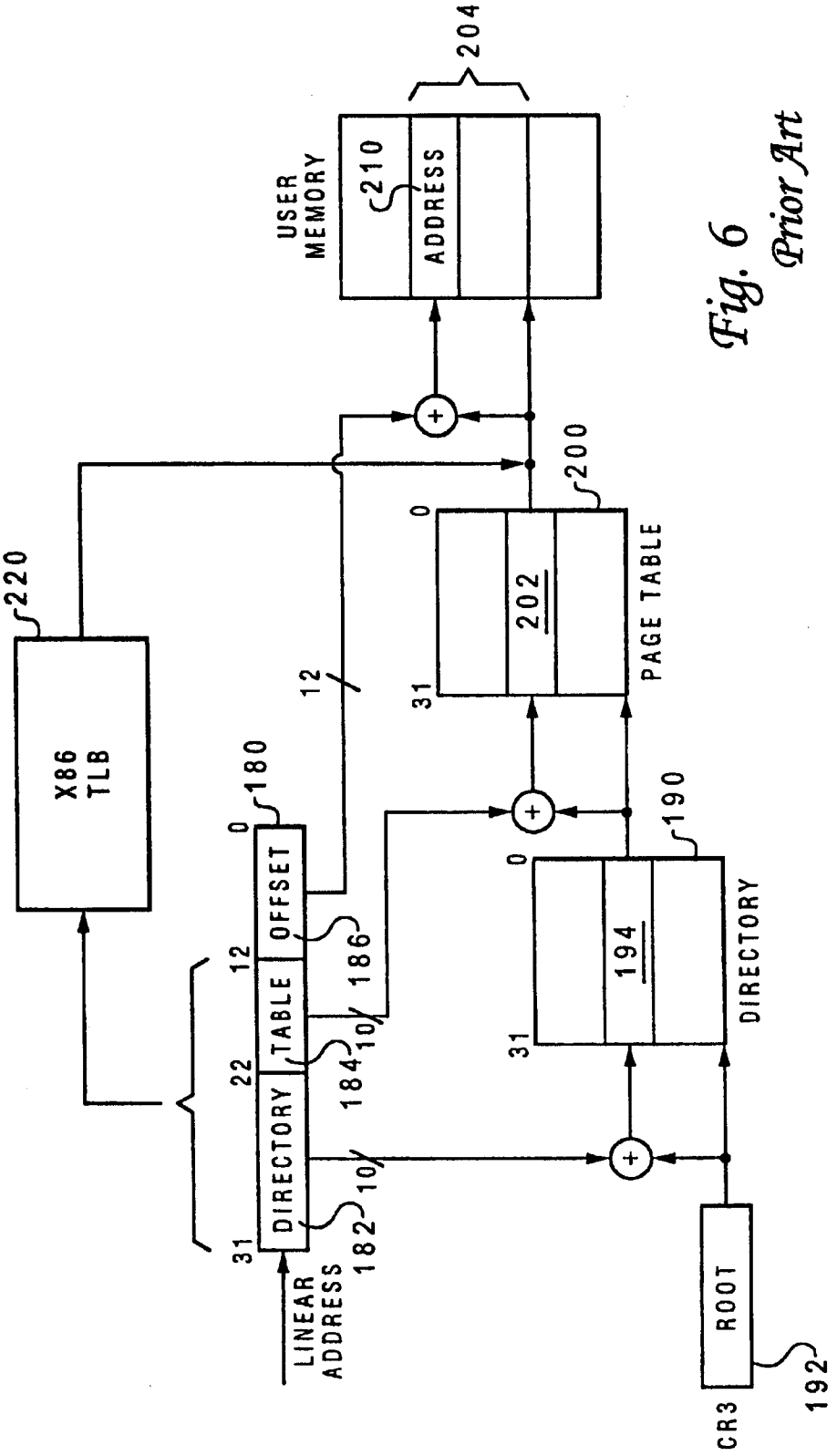



Fig. 6 Prior Art

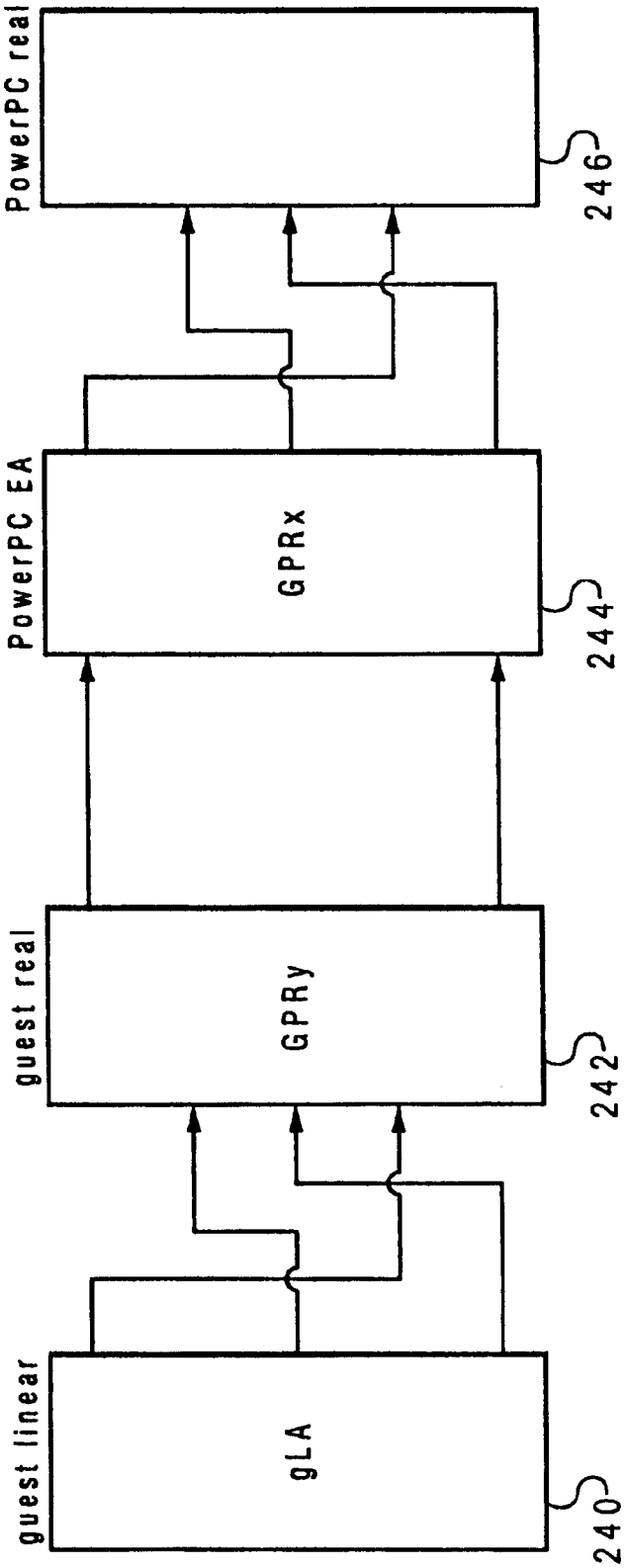

PowerPC real
246
PowerPC EA
GPRx
244
guest real
GPRy
242
guest linear
gLA
240

Fig. 7

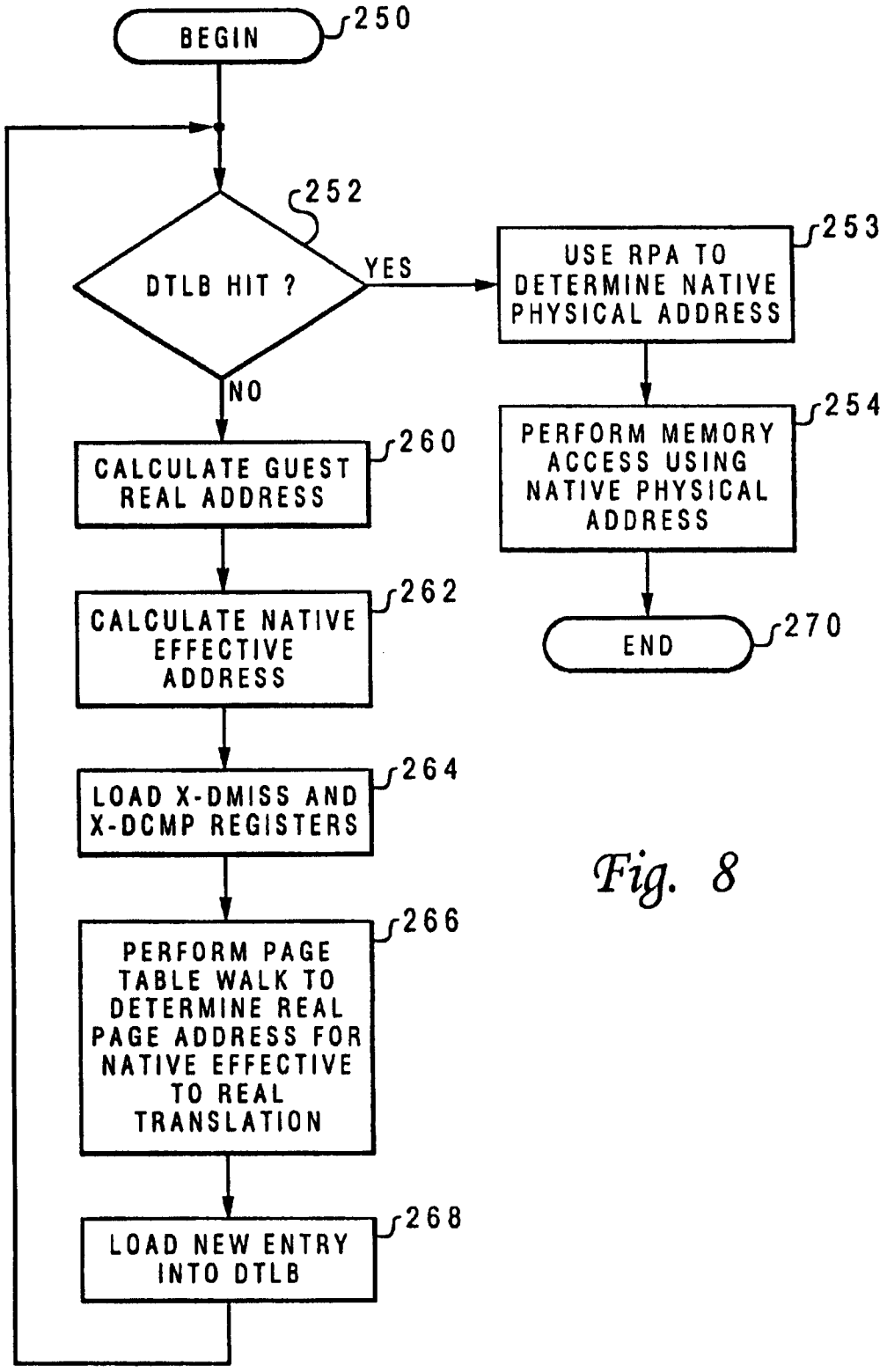

BEGIN
250
252
DTLB HIT ?
YES
NO
253
USE RPA TO DETERMINE NATIVE PHYSICAL ADDRESS
254
PERFORM MEMORY ACCESS USING NATIVE PHYSICAL ADDRESS
270
END
260
CALCULATE GUEST REAL ADDRESS
262
CALCULATE NATIVE EFFECTIVE ADDRESS
264
LOAD X-DMISS AND X-DCMP REGISTERS
266
PERFORM PAGE TABLE WALK TO DETERMINE REAL PAGE ADDRESS FOR NATIVE EFFECTIVE TO REAL TRANSLATION
268
LOAD NEW ENTRY INTO DTLB

*Fig. 8*

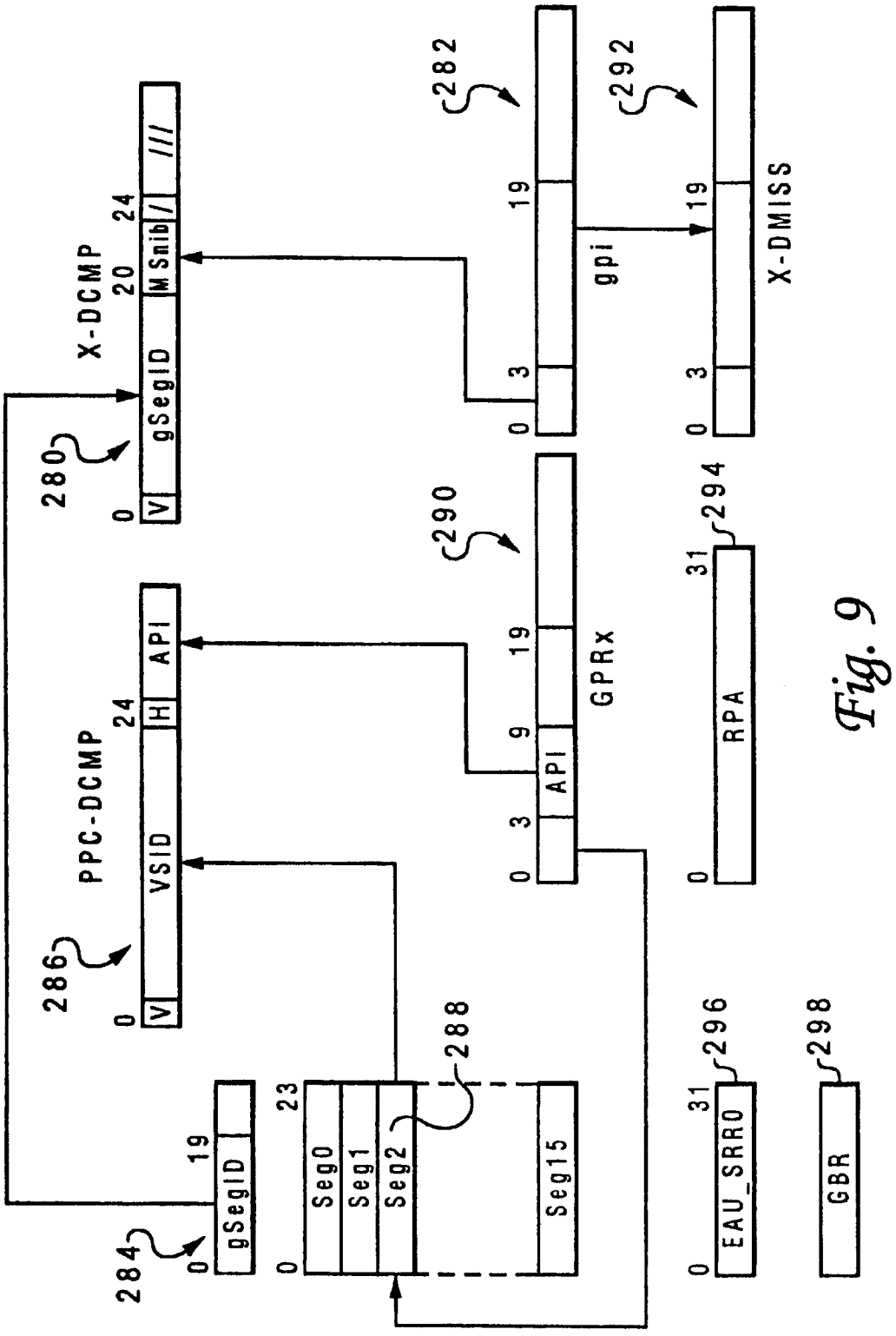

X-DCMP
V
gSegID
MSnib
///
0
20
24
280
PPC-DCMP
V
VSID
H
API
0
24
286
gSegID
0
19
284
Seg0
Seg1
Seg2
Seg15
0
23
288
282
gpi
0
3
19
292
X-DMISS
0
3
19
290
GPRx
API
0
3
9
19
294
RPA
0
31
296
EAU_SRR0
0
31
298
GBR

Fig. 9

# ADDRESS TRANSLATION BUFFER FOR DATA PROCESSING SYSTEM EMULATION MODE

## CROSS-REFERENCE TO RELATED APPLICATIONS

The subject matter of this application is related to that disclosed in the following applications, which are assigned to the assignee of the present application and are incorporated herein by reference:

Ser. No. (08,934,644) now U.S. Pat. No. 5,870,575, filed of even date herewith, for INDIRECT UNCONDITIONAL BRANCHES IN DATA PROCESSING SYSTEM EMULATION MODE, by James A. Kahle and Soummya Mallick.

Ser. No. 08,934,857, filed of even date herewith, for METHOD AND SYSTEM FOR PROCESSING BRANCH INSTRUCTIONS DURING EMULATION IN A DATA PROCESSING SYSTEM, by James A. Kahle and Soummya Mallick.

Ser. No. 08,935,007, filed of even date herewith, for METHOD AND SYSTEM FOR INTERRUPT HANDLING DURING EMULATION IN A DATA PROCESSING SYSTEM, by James A. Kahle and Soummya Mallick.

Ser. No. 08/591,291, filed Jan. 25, 1996, for A METHOD AND SYSTEM FOR MINIMIZING THE NUMBER OF CYCLES REQUIRED TO EXECUTE SEMANTIC ROUTINES, by Soummya Mallick.

Ser. No. 08/581,793, filed Jan. 25, 1996, for A METHOD AND SYSTEM FOR IMPROVING EMULATION PERFORMANCE BY PROVIDING INSTRUCTIONS THAT OPERATE ON SPECIAL-PURPOSE REGISTER CONTENTS, by Soummya Mallick.

## BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to a method and system for data processing and, in particular, to a method and system for emulating differing architectures in a data processing system. Still more particularly, the present invention relates to a method and system for address translation during emulation of guest instructions in a data processing system.

2. Description of the Related Art

The PowerPC™ architecture is a high-performance reduced instruction set (RISC) processor architecture that provides a definition of the instruction set, registers, addressing modes, and the like, for a family of computer systems. The PowerPC™ architecture is somewhat independent of the particular construction of the microprocessor chips or chips utilized to implement an instance of the architecture and has accordingly been constructed in various implementations, including the PowerPC 601™, 602™, 603™, and 604™. The design and operation of these processors have been described in published manuals such as the *PowerPC 604™ RISC Microprocessor User's Manual*, which is available from IBM Microelectronics as Order No. MPR604UMU-01 and is incorporated herein by reference.

As is true for many contemporary processors, a RISC architecture was chosen for the PowerPC™ because of the inherently higher performance potential of RISC architectures compared to CISC (complex instruction set computer) architectures. While it is desirable to optimize the design of a RISC processor to maximize the performance of the processor when executing native RISC instructions, it is also desirable to promote compatibility by accommodating commercial software written for CISC processors such as the Intel x86 and Motorola 68K.



Accordingly, an emulator mechanism can be incorporated into a PowerPC™ processor as disclosed in above-referenced Ser. No. 08/591,291 now U.S. Pat. No. 5,732,235 and Ser. No. 08/581,793 now U.S. Pat. No. 5,758,140. The disclosed emulation mechanism allows guest instructions (e.g., variable-length CISC instructions) to be emulated by executing corresponding semantic routines formed from native RISC instructions. Thus, the processor is required to manage two distinct instruction streams: a guest instruction stream containing the instructions to be emulated and a native instruction stream containing the native instructions within the semantic routines utilized to emulate the guest instructions. In order to maintain high performance when emulating guest instructions, an efficient mechanism is needed within the processor for managing both the guest and native instruction steams, with provision for branching, address translation buffer management, and exception handling.

The architecture of the Intel x86 line of microprocessors is described in *Microprocessors. Vol. I and Vol. II,* 1993, published by Intel Corporation as Publ. No. 230843. The Intel x86 instruction set is characterized in that the instructions are of variable length, from one byte in length to several bytes, and that arithmetic and logic operations can include a memory access (i.e., the operations can be memory-to-memory operations). In addition, complex addressing modes such as memory indirect are allowed. The architecture of the Motorola 68K line of microprocessors, which is described in various published documents such as *MC68030—Enhanced 32-bit Microprocessor User's Manual*, Prentice Hall, 1990, similarly uses complex addressing modes and variable-length instructions that can specify memory-to-memory operations.

The differences between RISC and CISC instruction sets also results in the utilization of diverse memory management methods and structures. For example, the Intel x86 architecture implements memory segmentation and paging in a manner that permits variable-length segments, while the PowerPC™ architecture employs fixed-length memory segments. Due to this and many other differences between the segmentation and paging mechanisms of the PowerPC and x86 architectures, the contents of page table and translation buffer entries are calculated using quite different logic.

Accordingly, the present invention includes the recognition that it would be desirable to provide a method and apparatus that permit a native (e.g., PowerPC™) architecture to use page table and translation buffer entries that are tailored to the memory management scheme of the guest (e.g., x86) instructions while performing the actual access to physical memory utilizing the native addressing mechanism.

## SUMMARY OF THE INVENTION

It is therefore one object of the present invention to provide an improved method and system for data processing.

It is another object of the present invention to provide a method and system for a method and system for emulating differing architectures in a data processing system.

It is yet another object of the present invention to provide a method and system for address translation during emulation of guest instructions in a data processing system.

The foregoing objects are achieved as is now described. According to one embodiment, an emulation mechanism for a host computer system allows guest instructions to be

executed by semantic routines made up of native instructions. The native instructions for the host processor are of a particular format, such as that specified by a RISC architecture, whereas the guest instructions are in a format for a different computer architecture, such as variable-length CISC instructions. The processor includes an emulator unit for fetching and processing the guest instructions that utilizes a multiple-entry queue to store the guest instructions currently fetched in order of receipt. Each entry in the queue includes an offset that indicates the location in memory of the semantic routine for the associated guest instruction, immediate data (if any) for the guest instruction, the length of the corresponding semantic routine, a condition field indicating results of arithmetic/logic operations by a guest instruction, valid bits, and other pertinent data. The processor executes a semantic routine in response to the entries in the queue, using the content of the entry to fetch the semantic routine. An entry is removed from the queue when the semantic routine for the associated guest instruction has been completed by the processor.

The memory management scheme for the guest instructions is different from that of the native instructions; accordingly, the translation of guest virtual addresses to guest real addresses is based on a different logic scheme. According to the present invention, a guest logical address is translated into a guest real address, which is thereafter translated into a native physical address. A semantic routine that emulates a guest instruction that accesses memory can then be executed utilizing the native physical address.

The above as well as additional objects, features, and advantages of the present invention will become apparent in the following detailed written description.

## BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. **1** depicts an illustrative embodiment of a data processing system with which the method and system of the present invention may advantageously be utilized;

FIG. **2** illustrates a more detailed block diagram of the processor depicted in FIG. **1**;

FIG. **3** depicts a more detailed block diagram of the emulation assist unit (EAU) in the processor of FIG. **2**;

FIG. **4** illustrates a more detailed block diagram of the guest instruction queue within the EAU depicted in FIG. **3**;

FIG. **5** is a diagram of a memory management scheme for CPU **4** utilized in the illustrative embodiment of FIG. **2**;

FIG. **6** is a diagram of a memory management scheme for the guest instructions used in the embodiment of FIG. **1–3**;

FIG. **7** is a diagram of an address translation scheme for guest instructions in the embodiment of FIGS. **1–6**;

FIG. **8** is a high level logical flowchart of a method for generating a guest TLB entry in accordance with the illustrative embodiment; and

FIG. **9** is a diagram of certain registers used in the address translation scheme illustrated in FIGS. **7** and **8**.

## DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

With reference now to the figures and in particular with reference to FIG. **1**, there is depicted a high level block diagram of a data processing system **2** in accordance with the present invention. As illustrated, data processing system **2**, which can comprise a desktop computer system, includes one or more CPUs **4**, which are connected to the other components of data processing system **2** in a standard hardware configuration. For example, CPUs **4** can be interconnected to main memory **6** via a memory controller **5** and system bus **7**. System bus **7** is also coupled to PCI (Peripheral Component Interconnect) bus **12** by a PCI host bridge **11**, which permits communication between the devices coupled to system bus **7** and peripherals **10** and I/O components **8**. Although for the purpose of illustration, the present invention is described below with reference to an illustrative embodiment in which CPU **4** is implemented with one of the PowerPC™ line of processors manufactured by International Business Machines Corporation, it should be understood that a variety of other processors could alternatively be employed.

When implemented as a PowerPC™ processor, each CPU **4** preferably comprises a single integrated circuit superscalar microprocessor, including various registers, buffers, execution units, and functional units that operate according to reduced instruction set computing (RISC) techniques. Each CPU **4** executes RISC instructions within the PowerPC™ instruction set architecture (e.g., the instructions forming application program **16** and operating system/kernel **18**) from a memory map **14**. The PowerPC™ instruction set architecture native to CPU **4** is defined in a number of publications such as *PowerPC™ User Instruction Set Architecture* and *PowerPC™ 603 RISC Microprocessor User's Manual* (Order No. MPR603UMU-01), both available from IBM Microelectronics. RISC instructions, such as those defined by the PowerPC™ instruction set architecture, can be characterized as having a fixed instruction length (e.g., 32-bits), including only register-to-register and register-to-memory operations and not memory-to-memory operations, and being executed without microcoding, often in one machine cycle.

Each CPU **4** is further adapted in accordance with the present invention to execute guest instructions (e.g., CISC instructions or some other instruction set that is not native to CPU **4**) by emulation. As described further hereinbelow, guest instructions **20** are each emulated by fetching and executing one or more semantic routines **19**, which each contain two or more native instructions. For example, a guest instruction **20** might be a memory-to-memory CISC instruction such as:

```
ADD MEM1, MEM2, MEM3
```

meaning "add the contents of memory location #1 to the contents of memory location #2 and store the result in memory location #3." A semantic routine **19** to emulate this guest CISC instruction might contain the following native RISC instructions:

```
LOAD REG1, MEM1
LOAD REG2, MEM2
ADD REG3, REG2, REG1
STORE REG3, MEM3
```

This exemplary semantic routine **19** loads the contents of memory locations #1 and #2 into registers #1 and #2, respectively, adds the contents of registers #1 and #2, stores the result of the addition in register #3, and stores the contents of register #3 to memory location #3. As further

illustrated in FIG. **1**, memory map **14** preferably further includes semantic routine synthesis code **22**, which comprises user-level code that can be utilized to synthesize a semantic routine corresponding to a guest instruction if such a semantic routine is not already one of the semantic routines in area **19**.

Referring now to FIG. **2**, there is illustrated a more detailed block diagram of CPU **4**. As depicted, CPU **4** is coupled to system bus **12** via a bus interface unit (BIU) **30** that controls the transfer of information between CPU **4** and other devices that are coupled to system bus **12**. BIU **30** is also connected to instruction cache **32** and data cache **34**. Both instruction cache **32** and data cache **34** are high-speed caches which enable CPU **4** to achieve a relatively fast access time to instructions and data previously transferred from main memory **6**, thus improving the speed of operation of data processing system **2**. Instruction cache **32** is further coupled to sequential fetcher **36**, which fetches native instructions from instruction cache **32** during each execution cycle. Sequential fetcher **36** transmits branch instructions fetched from instruction cache **32** to branch processing unit (BPU) **38** for execution, but temporarily buffers sequential instructions within instruction queue **40**. The sequential instructions stored within instruction queue **40** are subsequently dispatched by dispatch unit **42** to the sequential execution circuitry of CPU **4**.

In the depicted illustrative embodiment, the sequential execution circuitry of CPU **4** includes three (or more) execution units, namely, fixed-point unit (FXU) **50**, load/store unit (LSU) **52**, and floating-point unit (FPU) **54**. Each of these three execution units can execute one or more classes of native instructions, and all execution units can operate concurrently during each processor cycle. For example, FXU **50** performs fixed-point mathematical operations such as addition, subtraction, ANDing, ORing, and XORing, utilizing source operands received from specified general purpose registers (GPRs) **60** or GPR rename buffers **62**. Following the execution of a fixed-point instruction, FXU **50** outputs the data results of the instruction to GPR rename buffers **62**, which provide temporary storage for the data results until the data results are written to at least one of the GPRs **60** during the writeback stage of instruction processing. Similarly, FPU **54** performs floating-point operations, such as floating-point multiplication and division, on source operands received from floating-point registers (FPRs) **64** or FPR rename buffers **66**. FPU **54** outputs data resulting from the execution of floating-point instructions to selected FPR rename buffers **66**, which temporarily store the data results until the data results are written to selected FPRs **64** during the writeback stage of instruction processing. As its name implies, LSU **52** executes floating-point and fixed-point instructions which either load data from memory (i.e., either data cache **34** or main memory **6**) into selected GPRs **60** or FPRs **64** or which store data from a selected one of GPRs **60**, GPR rename buffers **62**, FPRs **64**, or FPR rename buffers **66** to data cache **34** or main memory **6**.

CPU **4** employs both pipelining and out-of-order execution of instructions to further improve the performance of its superscalar architecture. Accordingly, multiple instructions can be simultaneously executed by BPU **38**, FXU **50**, LSU **52**, and FPU **54** in any order as long as data dependencies and antidependencies are observed between sequential instructions. In addition, instructions are processed by each of FXU **50**, LSU **52**, and FPU **54** at a sequence of pipeline stages, including fetch, decode/dispatch, execute, finish and completion/writeback. Those skilled in the art should appreciate, however, that some pipeline stages can be reduced or combined in certain design implementations.

During the fetch stage, sequential fetcher **36** retrieves one or more native instructions associated with one or more memory addresses from instruction cache **32**. As noted above, sequential instructions fetched from instruction cache **32** are stored by sequential fetcher **36** within instruction queue **40**. In contrast, sequential fetcher **36** removes (folds out) branch instructions from the instruction stream and forwards them to BPU **38** for execution. BPU **38** preferably includes a branch prediction mechanism, which in an illustrative embodiment comprises a dynamic prediction mechanism such as a branch history table, that enables BPU **38** to speculatively execute unresolved conditional branch instructions by predicting whether or not the branch will be taken.

During the decode/dispatch stage, dispatch unit **42** decodes and dispatches one or more native instructions from instruction queue **40** to an appropriate one of sequential execution unit **50**, **52**, and **54** as dispatch-dependent execution resources become available. These execution resources, which are allocated by dispatch unit **42**, include a rename buffer within GPR rename buffers **60** or FPR rename buffers **66** for the data result of each dispatched instruction and an entry in the completion buffer of completion unit **44**.

During the execute stage, execution units **50**, **52**, and **54** execute native instructions received from dispatch unit **42** opportunistically as operands and execution resources for the indicated operations become available. In order to minimize dispatch stalls, each one of the execution units **50**, **52**, and **54** is preferably equipped with a reservation table that stores dispatched instructions for which operands or execution resources are unavailable.

After the operation indicated by a native instruction has been performed, the data results of the operation are stored by execution units **50**, **52**, and **54** within either GPR rename buffers **62** or FPR rename buffers **66**, depending upon the instruction type. Then, execution units **50**, **52**, and **54** signal completion unit **44** that the execution unit has finished an instruction. In response to receipt of a finish signal, completion unit **44** marks the completion buffer entry of the instruction specified by the finish signal as complete. Instructions marked as complete thereafter enter the write-back stage, in which instructions results are written to the architected state by transferring the data results from GPR rename buffers **62** to GPRs **60** or FPR rename buffers **66** to FPRs **64**, respectively. In order to support precise exception handling, native instructions are written back in program order.

As illustrated in FIG. **2**, in order to facilitate the emulation of guest instructions, CPU **4** includes emulation assist unit (EAU) **70**, which is shown in greater detail in FIG. **3**. As illustrated in FIG. **3**, EAU **70** includes a number of special purpose registers (SPRs) **76** for storing, among other things, the logical base address of segments of guest address space containing guest instructions. SPRs **76** include a code segment base register (CSBR) **75** that stores the base address of the current segment and an offset to the current guest instruction. EAU **70** further includes an instruction prefetch unit **74** for fetching guest instructions from data cache **34** and an instruction buffer **72** for temporarily storing guest instructions retrieved from data cache **34**. In addition, EAU **70** includes an instruction decoder **78** for decoding guest instructions, a guest branch unit **80** for executing guest branch instructions, tag generation unit **90**, which generates opcode tags for each sequential guest instruction, guest cache **92**, which stores a semantic routine (SR) offset in association with each of a plurality of opcode tags, a guest

instruction queue **100** for storing information associated with guest instructions, and a guest dispatch unit **94** that provides SR addresses to sequential fetcher **36**.

Referring now to FIG. **4**, there is illustrated a more detailed view of guest instruction queue **100**, which provides a synchronization point between the guest instruction stream and native instruction stream. As will become apparent from the following description, the provision of guest instruction queue **100** permits guest instructions emulated by CPU **4** to be pre-processed so that the latency associated with the various emulation pipeline stages can be overlapped.

In the illustrative embodiment, guest instruction queue **100** contains five entries **110**, which each include the following fields **112–134**:

- $V_I$: indicates whether the content of immediate field **116** is valid
- $V_O$: indicates whether the content of SR offset field **118** is valid
- Immediate: stores immediate data that is specified by the guest instruction and is passed as a parameter to the corresponding semantic routine
- SR offset: offset between the base address of the guest instruction (which is maintained in CSBR **75**) and the corresponding semantic routine
- L: length of semantic routine in native instructions
- GIP: offset pointer from CSBR **75** to guest instruction in guest address space
- DE: indicates whether two guest instruction queue entries (and two semantic routines) are utilized in the emulation of a single guest instruction
- S: indicates whether the guest instruction is in a speculative (i.e., predicted) execution path in the guest instruction stream
- BBID: unique basic block ID number sequentially assigned to each semantic routine from pool of BBIDs
- GCR: guest condition register that indicates conditions (e.g., equal/not equal) that may be utilized to predict subsequent guest branch instructions
- CRM: guest condition register mask that indicates which bits in the GCR field will be altered by the guest instruction
- $V_B$: indicates whether the semantic routine native instruction that will set the value of GCR field **130** has executed

As depicted in FIG. **4**, guest instruction queue **100** has an associated emulation instruction pointer register (EIPR) **140**, preferably implemented as a software-accessible special purpose register (SPR), which contains the offset from the base address specified by CSBR **75** to the current guest instruction that is being interpreted. EAU **70** updates the contents of EIPR **140** in response to the execution of a newly-defined "guest dispatch completion" (gdispc) instruction in the native instruction set and in response to the execution of a guest branch instruction by guest branch unit **80** without invoking a semantic routine. Another special purpose register, emulation instruction pointer next (EIP_NEXT) register **142**, contains the offset from the base address specified in CSBR **75** to the next guest instruction that will be interpreted. EAU **70** updates the contents of EIP_NEXT register **142** when a gdispc instruction is executed, when a special move to SPR instruction (i.e., mtspr[EIP_NEXT]) is executed having EIP_NEXT register **142** as a target, and when a guest branch or guest NOOP instruction is emulated without invoking a semantic routine. These two offset pointers permit the state of the guest instruction stream to be easily restored following a context switch, for example, when returning from an exception. That is, by saving both the current EIP and the next EIP, the guest instruction under emulation at the time of the interrupt, which is pointed to by the current EIP, does not need to be reexecuted to compute the next EIP if both the current EIP and next EIP are saved.

Guest instruction queue **100** also has an associated predicted taken flag (PTF) **144**, which indicates whether an unresolved guest branch instruction was predicted as taken and therefore whether sequential guest instructions marked as speculative (i.e., S field **126** is set) are within the target or sequential execution path.

Xflags **150** is an architected condition register for which GCR **130** in each of entries **110** is a "renamed" version. When an entry **110** is removed from the bottom of guest instruction queue **100**, the bits within Xflags **150** specified by CRM **132** in that entry **110** are updated by the corresponding bit values in GCR **130**. Xflags **150**, GCR fields **130**, CRM fields **132**, and $V_B$ fields **134** (and the associated access circuitry), which are identified in FIG. **3** simply as guest flags **83**, can be referenced by guest branch unit **80** to resolve guest branch instructions as described further herein below.

In cases in which each guest instruction is emulated by executing a single semantic routine, each guest instruction is allocated only a single entry **110** within guest instruction queue **100**. However, in some circumstances more than one entry **110** may be allocated to a single sequential guest instruction. For example, in an embodiment in which the guest instructions are x86 instructions, many sequential guest instruction comprise two distinct portions: a first portion that specifies how the addresses of the source(s) and destination of the data are determined and a second portion that specifies the operation to be performed on the data. In such cases, a first semantic routine is utilized to emulate the portion of instruction execution related to the determination of the data source and destination addresses and a second semantic routine is utilized to emulate the portion of instruction execution related to performing an operation on the data. Accordingly, the guest instruction is allocated two entries **110** in guest instruction queue **100**—a first entry containing information relevant to the first semantic routine and a second entry containing information relevant to the second semantic routine. Such dual entry guest instructions are indicated within guest instruction queue **100** by setting DE (dual entry) field **124** in the older (first) of the two entries **110**. Setting the DE field ensures that both entries **110** will be retired from guest instruction queue **100** when both semantic routines have completed (i.e., in response to a gdispc instruction terminating the second semantic routine). The emulation of guest instructions utilizing two semantic routines advantageously permits some semantic routines to be shared by multiple guest instructions, thereby reducing the overall memory footprint of semantic routines **19**.

The ordering of the entries **110** in guest instruction queue **100** is maintained by current entry pointer **95**, which points to the oldest entry in guest instruction queue **100**, and next entry pointer **97**, which points to the next oldest entry. In response to a fetch or completion of a gdispc instruction, the guest instruction queue entry indicated by current entry pointer **95** is retired and both current entry pointer **95** and next entry pointer **97** are updated. Thus, entries are consumed from the "bottom" and inserted at the "top" of guest instruction queue **100**.

With reference now to FIGS. **2–4**, the operation of EAU **70** will now be described.

EAU INITIALIZATION

To initialize EAU **70** for emulation, the address offset to the first guest instruction to be emulated is loaded into EIP_NEXT register **142** by executing a native move to SPR (mtspr) instruction having EIP_NEXT register **142** as a target (i.e., mtspr[EIP_NEXT] in the PowerPC™ instruction set). In a preferred embodiment, this native instruction is equivalent to a guest branch always instruction since the function of such a guest branch instruction would be to load EIP_NEXT register **142** with a pointer to the next guest instruction to be executed (i.e., the offset value within CSBR **75**). $V_I$ field **112** and $V_O$ field **114** of the oldest entry **110** in guest instruction queue **100** are both cleared in response to the mtspr[EIP_NEXT] instruction. Thereafter, prefetching of guest instruction from data cache **34** can be triggered utilizing a gdispc instruction.

As an aside, $V_I$ field **112** and $V_O$ field **114** of the oldest entry **110** in guest instruction queue **100** are also cleared in response to mtspr[EIP] and mtspr[CSBR] instructions, as well as when a guest branch instruction is resolved as mispredicted.

GUEST INSTRUCTION PREFETCHING

As noted above, prefetching of guest instructions from data cache **34** is triggered by placing a gdispc instruction in the native instruction stream. When fetched by sequential fetcher **36**, the gdispc instruction acts as an interlock that stalls fetching by sequential fetcher **36** until $V_O$ field **114** of the oldest entry **110** in guest instruction queue **100** is set. In response to the stall of sequential fetcher **36**, instruction prefetch unit **74** in EAU **70** makes a fetch request to data cache **34** for the guest instruction at the address specified by the base address and offset contained in CSBR **75**.

GUEST INSTRUCTION DECODING

Guest instructions supplied by data cache **34** in response to fetch requests from instruction prefetch unit **74** are temporarily stored in instruction buffer **72** and then loaded one at a time into instruction decoder **78**, which at least partially decodes each guest instruction to determine the instruction length, whether the guest instruction is a branch instruction, and the immediate data of the guest instruction, if any.

GUEST BRANCH INSTRUCTION PROCESSING

If instruction decoder **78** determines that a guest instruction is a branch instruction, the guest branch instruction is forwarded to guest branch unit **80** for processing after allocating the guest branch instruction the oldest unused entry **110** of guest instruction queue **100**. (In an alternative embodiment, guest instruction ordering can be maintained without assigning guest instruction queue entries to guest branch instructions). Guest branch unit **80** first attempts to resolve a conditional guest branch instruction with reference to guest flags **83**. If the bit(s) upon which the guest branch depends are set within CRM field **132** and $V_B$ field **134** is marked valid in the entry **110** corresponding to the immediately preceding sequential guest instruction, guest branch unit **80** resolves the guest branch instruction by reference to GCR field **130** of the entry **110** of immediately preceding sequential guest instruction. If, however, the relevant bit(s) within CRM **132** in the entry **110** corresponding to the immediately preceding sequential guest instruction are not set, the guest branch instruction is resolved by reference to the newest preceding entry **110**, if any, having the relevant bits set in CRM field **132** and $V_B$ field **134** marked as valid, or failing that, by reference to Xflags **150**. On the other hand, guest branch unit **80** predicts (i.e., speculatively executes) a conditional guest branch instruction by reference to conventional branch history table (BHT) **82** if $V_B$ field **134** is marked invalid in the newest preceding entry **110** in which the bit(s) relevant to the guest branch are set in GCR field **130**. The guest instruction at the address of the resolved or predicted execution path is thereafter fetched from data cache **34** via instruction prefetch unit **74**. Further details about the processing of guest branch instructions are found in Ser. No. 08,934,644 now U.S. Pat. No. 5,870,575 and Ser. No. 08/934,857, which were referenced hereinabove.

SEQUENTIAL GUEST INSTRUCTION PROCESSING

If the guest instruction decoded by instruction decoder **78** is a sequential instruction, at least the oldest unused entry **110** of guest instruction queue **100** is allocated to the guest instruction. As illustrated in FIG. **3**, instruction decoder **78** then stores the immediate data, if any, and the offset pointer to the guest instruction into immediate field **116** and GIP field **122**, respectively, of the allocated entry **110**. In response to instruction decoder **78** loading immediate data into immediate field **116**, $V_I$ field **112** is set.

The sequential guest instruction is then forwarded from instruction decoder **78** to tag generation unit **90**, which converts the guest instruction into a unique opcode tag. According to a preferred embodiment, different opcode tags are utilized not only to distinguish between different guest instructions, but also to distinguish between identical guest instructions that access different registers. Thus, different opcode tags are utilized for guest divide (gdiv) and guest multiply (gmult) instructions, as well for gmult R3,R2,R1 and gmult R4,R2,R1 instructions, which target different registers. The unique opcode tag produced by tag generation unit **90** forms an index into guest cache **92** that selects a particular cache entry containing an offset utilized to determine the effective address of the semantic routine corresponding to the guest instruction.

As indicated, in the illustrative embodiment, guest cache **92** comprises a four-way set associative cache having 256 lines that each contain four 4 Kbyte entries. A miss in guest cache **92** generates a user-level interrupt, which is serviced by executing semantic routine synthesis code **22**. As described above, semantic routine synthesis code **22** synthesizes a semantic routine corresponding to the guest instruction from native instructions and stores the semantic routine in area **19** of memory map **14**. The offset from the base address of the guest instruction to the location of the newly synthesized semantic routine is then stored in guest cache **92** for subsequent recall. Because guest instruction sets are typically fairly stable, it is typical for guest cache **92** to achieve hit rates above 99%.

In response to the semantic routine (SR) offset being located (or stored) in guest cache **92**, the SR offset is stored in SR offset field **118** of the allocated entry **110**, thereby causing $V_O$ field **114** to be marked as valid. By the time $V_O$ is set to signify that the content of SR offset field **118** is valid, L field **120**, DE field **124**, S field **126**, BBID field **128**, and CRM field **132** are also valid within the allocated entry **110**. As noted above, GCR field **130** is indicated as valid separately by $V_B$ field **134**.

When $V_O$ field **114** of the oldest entry **110** in guest instruction queue **100** is set by the processing of the first guest instruction in EAU **70** at emulation startup, the value in EIP_NEXT register **142** is transferred to EIPR **140**, signifying that the oldest (i.e., first) instruction in guest instruction queue **100** is the guest instruction currently being processed. In response to this event, guest dispatch unit **94** transmits the SR offset in SR offset field **118** to sequential fetcher **36**, which begins to fetch native instructions within the semantic routine corresponding to the first guest instruction. As illustrated in FIG. **4**, EAU **70** tracks the guest

instruction for which the semantic routine is being fetched utilizing fetch PTR **152** in guest dispatch unit **94**.

SEMANTIC ROUTINE PROCESSING

Semantic routine (i.e., native) instructions that are within the standard instruction set of CPU **4** are processed by CPU **4** as described above with reference to FIG. **2**. Special instructions inserted into the native instruction set to support guest instruction emulation are handled as described below.

In order to connect guest instructions into a continuous guest instruction stream, a gdispc instruction is preferably inserted at the end of each semantic routine, if the guest instructions are each represented by a single semantic routine, or at the end of the last semantic routine corresponding to the guest instruction, if the guest instruction is emulated by multiple semantic routines. The gdispc instruction is preferably defined as a special form of a native branch instruction so that when fetched from instruction cache **32** by sequential fetcher **36** a gdispc instruction is folded out of the native instruction stream and passed to BPU **38**. In response to detecting the gdispc instruction, BPU **38** asserts signal line **96**. Guest dispatch unit **94** responds to the assertion of signal line **96** by removing all of the entries **110** corresponding to the current guest instruction from guest instruction queue **100** and by passing the semantic routine offset stored within the next entry to sequential fetcher **36** via signal lines **98**. As described above, sequential fetcher **36** then computes the effective address (EA) of the semantic routine corresponding to the next guest instruction by adding the semantic routine offset to the guest instruction's base address and fetches the semantic routine from memory for execution by CPU **4**.

When multiple semantic routines are utilized to emulate a single guest instruction, semantic routines other than the final semantic routine are terminated by a "guest dispatch prolog completion" (gdispp) instruction, which is a variant of the gdispc instruction. In general, the gdispp instruction is processed like the gdispc instruction. For example, like the gdispc instruction, the gdispp instruction triggers the fetching of the next semantic routine. In addition, $V_O$ field **114** within the guest instruction queue entry **110** corresponding to the semantic routine containing a gdispp instruction must be set in order for the gdispp instruction to be executed. However, in contrast to the processing of a gdispc instruction, the completion of a gdispp instruction does not trigger the removal of an entry **110** from guest instruction queue **100** or the updating of EIPR **140** and EIP_NEXT register **142**.

Another special instruction inserted into the native instruction set as a form of add instruction is the guest add immediate prolog [word or half word] (gaddpi[w,h]) instruction. The function of the gaddpi[w,h] instruction is to add the immediate data specified in the first of two guest instruction queue entries allocated to a guest instruction with the value in a specified GPR **60** and store the sum in another GPR **60**. Accordingly, $V_I$ field **112** for the first entry **110** must be set in order to permit the corresponding semantic routine to execute.

A similar guest add immediate completion [word or half word] (gaddci[w,h]) instruction is utilized to add the immediate data stored in the second of two guest instruction queue entries allocated to a guest instruction with value of a specified GPR **60** and store the sum in another GPR **60**. $V_I$ field **112** for the second entry **110** must be set in order for the corresponding semantic routine to execute.

INTERRUPT AND EXCEPTION HANDLING

In response to either a guest instruction or native instruction exception, a non-architected exception flag is set that disables guest instruction fetching by instruction prefetch unit **74**. At a minimum, the context of the guest instruction stream is saved during interrupt/exception handling and restored upon returning from the interrupt/exception by saving the contents of EIPR **140** and EIP_NEXT register **142** in SPRs **76**. As a practical matter, it is preferable to save the entire bottom entry **110** of guest instruction queue **100** in SPRs **76** in order to expedite the restart of emulation following the interrupt/exception.

Prefetching of guest instructions from data cache **34** following a return from interrupt can be triggered by the execution of either a gaddpi[w,h] instruction or gaddci[w,h] instruction, which interlocks with and stalls sequential fetcher **36** until $V_I$ field **112** of the appropriate entry **110** in guest instruction queue **100** is set. Guest instruction prefetching may also be restarted through the execution of a gdispc instruction or gdispp instruction. The execution of a gdisp[p,c] or gadd[p,c]i[w,h] instruction clears the exception flag.

MEMORY MANAGEMENT

Referring again to FIG. **2**, instruction cache **32** (and main memory **6**) is accessed via an instruction memory management unit (IMMU) **58**, and likewise data cache **34** (and main memory **6**) is accessed via a data memory management unit (DMMU) **56**. Each of memory management units **56** and **58** has its own respective translation lookaside buffer, so there is an ITLB **59** and a separate DTLB **57**. TLBs **57** and **59** each contain copies of page table entries from the page tables in memory **6**, which correlate real (physical) addresses of pages in memory with logical (effective) addresses generated by CPU **4** for instructions and data. Since the memory management scheme for the guest (e.g., Intel x86) instruction architecture may differ significantly from the native PowerPC™ scheme, memory management units (MMUs) **56** and **58** include address translation facilities for guest instructions that reference memory.

Referring now to FIG. **5**, there is depicted a diagram of the scheme utilized by MMUs **56** and **58** to translate logical (effective) addresses into physical addresses while processing native instructions. As illustrated, in response to receipt of a 32-bit logical (effective) address **60** by one of MMUs **56** and **58**, a determination is made whether address translation is enabled. If not, the logical address is utilized as the physical address, as indicated at reference numeral **174**. If, however, address translation is enabled, the MMU utilizes 4 high order bits from the logical address to select a segment register that contains a 24-bit segment address, as depicted at reference numeral **162**. In parallel with the selection of the segment register, the logical address is compared with the address ranges defined in a Block Address Translation (BAT) array as illustrated at reference numeral **164**. If the logical address falls with an address range defined in the BAT array, then block address translation is performed to obtain a 32-bit physical address as indicated at reference numeral **166**.

However, if no match is found for the logical address in the BAT array, a control bit in the descriptor of the selected segment is tested to determine if the access is to memory or to I/O controller interface space. If the bit in the segment descriptor indicates that the access is to I/O controller interface space, I/O controller interface translation is performed to obtain a 32-bit I/O controller interface address, as shown at reference numeral **168**. Otherwise, page address translation is performed by concatenating the 24-bit segment address with the low order 28 bits of the logical address to produce a 52-bit virtual address **170**, and by thereafter translating this 52-bit virtual address into a 32-bit physical

address, if possible, by reference to a page table entry (as indicated at reference numeral **172**). If the required page table entry (PTE) is present in the relevant one of DTLB **57** and ITLB **59**, the physical address corresponding to the logical address is immediately available. However, if a TLB miss occurs, an exception is taken, and the page table in memory is searched for the matching PTE.

Regardless of whether page address translation, block address translation, or direct address translation is performed, the resulting 32-bit physical address can then be utilized to access one of instruction cache **32** and data cache **34**, and if a cache miss occurs, main memory **6**.

With reference now to FIG. **6**, the conventional two-level address translation scheme of the Intel x86 architecture is illustrated in diagram form. As depicted, a 32-bit linear address **180** is translated into a physical address by first partitioning linear address **180** into a 10-bit directory field **182**, a 10-bit table field **184**, and a 12-bit offset field **186**. The value of directory field **182** is utilized as an offset that, when added to a root address stored in control register **192**, accesses an entry **194** in page directory **190**. Page directory entry **194** contains a pointer that identifies the base address of page table **200**. The value of table field **184** forms an offset pointer that, when added to the value of directory entry **194**, selects a page table entry **202** that specifies the base address of a page **204** in memory. The value of offset field **186** then specifies a particular physical address **210** within page **204**.

As depicted in FIG. **6**, the 20 high order bits of linear address **180** are also utilized in parallel to search for a matching page table entry in x86 TLB **220**. If a match is found in x86 TLB **220**, the matching page table entry is utilized to perform linear-to-real address translation in lieu of page directory **190** and page table **200**, which require memory accesses.

By comparison of FIGS. **5** and **6**, it should be apparent that guest instructions require a different page address translation scheme than native instructions. Accordingly, referring now to FIG. **7**, a high level diagram of the method employed by the present invention to translate guest linear addresses into native real (physical) addresses is shown. According to the present invention, each guest linear address **240**, which may be the target address of a guest instruction fetch, guest data load, or guest data store, is first translated into a guest real address **242** using the logical equivalent of the address translation scheme depicted in FIG. **6**. A simple mapping translation, such as adding a fixed offset (which may be zero), is then performed to obtain native effective address **244**. Subsequently, a native physical (real) address **246** is calculated using the logical equivalent of the process illustrated in FIG. **5**.

With reference now to FIG. **8**, a high level logical flowchart is provided that illustrates how the process of guest address translation shown in FIG. **7** is preferably implemented within CPU **4**. In the preferred embodiment, all guest storage accesses (i.e., guest instruction fetches, data loads, and data stores) are treated as data accesses by CPU **4**, and are accordingly accessed using DMMU **56**. The registers utilized by DMMU **56** to translate guest addresses are illustrated in FIG. **9**.

As depicted in FIG. **8**, the process of guest address translation begins at block **250** in response to receipt of a guest linear address by DMMU **56**. DMMU **56** distinguishes between guest linear addresses and native effective addresses received as inputs by the presence (or absence) of a bit generated when the memory access instruction was decoded. The process proceeds from block **250** to block **252**, which illustrates a determination of whether or not DTLB **57** contains a "guest" entry that maps the guest linear address to a native physical address. In a preferred embodiment, DTLB **57** contains both native entries as well as guest entries, which are marked, for example, with an "X" bit set to one. However, in other embodiments, DTLB **57** may include separate TLBs for native and guest entries. In either case, the determination depicted at block **252** may be made, for example, by comparing the 32-bit guest linear address with the first 32 bits of each guest entry. In response to a hit in DTLB **57**, the process proceeds from block **252** to block **253**, which illustrates DMMU **56** utilizing the native real page address (described below) in the matching DTLB entry to calculate a native physical address. Next, as shown at block **254**, DMMU **56** utilizes the native physical address to access the requested data within either data cache **34** or main memory **6**. Thereafter, the process terminates at block **270**.

Referring again to block **252**, in response to a miss in DTLB **57**, a guest DTLB miss exception is generated. Before branching to appropriate exception handler, hardware within CPU **4** saves the address of the semantic routine native instruction at which the DTLB miss exception occurred in one of SPRs **76** designated as EAU_SRRO **296**. In a preferred embodiment, this native instruction address is contained in a completion buffer entry within completion unit **44**. In addition, hardware saves the 20 high order bits of the guest linear address **180** that caused the DTLB miss in one of SPRs **76** designated as guest page index (GPI) register **282**. Execution by CPU **4** then branches to a user-level exception handler located at a predetermined offset from the value stored in guest branch register (GBR) **298**. In order to simplify the exception handling logic, the offset of the guest DTLB miss exception handler is preferably equal to the offset assigned to the native DTLB miss exception handler.

As illustrated at block **260** of FIG. **8**, the guest DTLB exception handler utilizes the logical equivalent of the address translation process depicted in FIG. **6** to generate a guest real address **242** from the guest linear address stored in GPI register **282**. The guest real address **242** is then stored within a general purpose register GPRy. As indicated at block **262**, a fixed offset value (which may be zero) is added to the guest real address to produce a native effective address, which is then stored in one of GPRs **60** designated as GPRx **290**.

The process then proceeds from block **262** to block **264**, which illustrates the execution of a "guest TLB load" (gtlbld) instruction in the exception handler routine. The gtlbld instruction is a newly defined operation in the native instruction set that builds the first word of a guest TLB entry. As illustrated in FIG. **9**, the execution of the gtlbld instruction causes hardware within CPU **4** to load the conventional PPC-DCMP register **286** with bits 4–9 (the abbreviated page index (API)) of GPRx **290** and with the value stored in the native segment register **288** selected by the four most significant bits of GPRx **290**. Thus, PPC-DCMP register **286** has the same format as the first word of a conventional native DTLB entry. The execution of the gtlbld instruction also causes the hardware of CPU **4** to load selected fields of (guest) X-DCMP register **280** with the 4 high order bits of GPI register **282** and with the 19-bit value of gSegID **284**, which, similar to the virtual page numbers stored in native segment registers **288**, specifies high order bits of a virtual page number assigned to guest instructions and data. The gtlbld instruction also loads X-DMISS register **292** with the content of GPI register **282**, which stores the 20 high order bits of the guest linear address **180** that caused the DTLB

miss. Finally, as illustrated at block **266** of FIG. **8**, execution of the gtlbld instruction generates a hardware exception that invokes the conventional PowerPC™ supervisor-level page table walk routine. The page table walk routine determines the PowerPC™ (native) real page address to which the guest linear address maps and creates a guest entry in DTLB **57** to translate the guest linear address.

The table walk routine begins by comparing the value of PPC-DCMP register **286** against the first word of page table entries in memory (data cache **34** and main memory **6**) until a match is found. In response to finding a matching entry in the page table, the supervisor-level table walk routine loads real page address (RPA) register **294** with the second word of the matching page table entry. A native TLB load (tlbld) instruction in the supervisor-level table walk routine is then executed. In response to the tlbld instruction, the hardware of CPU **4** selects an entry within DTLB **57**, which preferably comprises a 32 entry 2-way set associative TLB array, utilizing bits 15–19 of X-DMISS register **292**. In the illustrative embodiment, each DTLB entry comprises a first set including 36 bits and a second set containing 32 bits. As illustrated at block **268** of FIG. **8**, the tlbld instruction then causes bits 1–23 of X-DCMP register **280** to be loaded into bits 2–24 of the first set of the selected DTLB entry, bits 4–14 of X-DMISS register **292** to be loaded into bits 25–35 of the first set of the selected DTLB entry, and the value of RPA register **294** to be loaded into the second set of the selected DTLB entry. To signify that the selected DTLB entry contains a guest address translation, the gtlbld instruction also sets an X bit, which in a preferred embodiment comprises bit 1 (i.e., the second most significant bit) of the first set of the selected DTLB entry. The supervisor-level table walk routine then executes a native "return from interrupt" (rfi) instruction and returns control to the user-level exception handler.

Thereafter, the user-level exception handler returns to the emulation of guest instructions utilizing the following sequence of native instructions:

```
mfspr GPRz, EAU_SRRO

mtctr GPRz

bcctr BO[0]=1
```

The mfspr ("move from SPR") instruction loads the effective address of the instruction at which the exception occurred from EAU_SRRO **296** into GPRz. The effective address is then transferred from GPRz into the PowerPC™ count register via the mtctr ("move to count register") instruction. The guest context of EAU **70** is then restored to its pre-exception state by executing the bcctr ("branch conditional to count register") instruction, which causes sequential fetcher **36** to resume fetching native instructions at the specified effective address (i.e., at the point where sequential fetcher **36** discontinued fetching native instructions in response to the DTLB miss exception). When the native instruction at which the exception occurred is again executed, the guest linear address is again passed to DMMU **56** for translation, as illustrated at block **252**. This time the guest linear address hits in DTLB **57**, and the process proceeds to block **253**. Block **253** depicts the formation of the native physical address by concatenating the 19 high order bits of the second set of the matching DTLB entry (i.e., the real page number) with the 12-bit offset of the guest linear address. As depicted at block **254**, DMMU **56** then performs the memory access utilizing the native real address. Thereafter, the process terminates at block **270**.

While an illustrative embodiment of the present invention has been particularly shown and described, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope hereof. For example, while the present invention has been described with reference to embodiments in which the guest instructions emulated within CPU **4** are x86 CISC instructions, it should be understood that other guest instructions could alternatively be emulated.

What is claimed is:

**1**. A method of operating a processor which has a native instruction set and emulates instructions in a guest instruction set, said method comprising:

storing, in memory, a series of guest instructions from said guest instruction set, said series including a guest memory access instruction that indicates a guest logical address in guest address space;

for each guest instruction in said series, storing in memory a semantic routine of native instructions from said native instruction set to emulate each guest instruction, said native instructions utilizing native addresses in native address space;

in response to receipt of said guest memory access instruction for emulation, translating said guest logical address into a guest real address and thereafter translating said guest real address into a native physical address; and

executing a semantic routine that emulates said guest memory access instruction utilizing said native physical address.

**2**. The method of claim **1**, wherein said guest memory access instruction comprises one of a guest load instruction and a guest store instruction.

**3**. The method of claim **1**, wherein said guest memory access instruction comprises a guest instruction that initiates fetching of a guest instruction in said series from memory.

**4**. The method of claim **1**, wherein said step of executing a semantic routine that emulates said guest memory access instruction comprises the step of accessing said memory utilizing said native physical address.

**5**. The method of claim **1**, said step of translating said guest real address into a native physical address includes the step of translating said guest real address into a native effective address and then translating said native effective address into said native physical address.

**6**. The method of claim **1**, said processor including a translation lookaside buffer (TLB) containing entries utilized for address translation, wherein said step of translating said guest logical address into a guest real address and thereafter translating said guest read address into a native physical address comprises the steps of:

determining if said translation lookaside buffer includes an entry that can be utilized to obtain said native physical address; and

in response to a determination that said translation lookaside buffer contains an entry that can be utilized to obtain said native physical address, translating said guest logical address into a guest real address and thereafter translating said guest read address into a native physical address utilizing said translation lookaside buffer (TLB) entry.

**7**. The method of claim **6**, said method further comprising the step of:

in response to a determination that said translation lookaside buffer does not contain an entry that can be utilized to obtain said native physical address, creating an entry

that can be utilized to obtain said native physical address in said translation lookaside buffer.

**8**. The method of claim **1**, wherein said translating step performed utilizing a user-level semantic routine.

**9**. A processor which has a native instruction set and emulates instructions in a guest instruction set, said processor comprising:

guest instruction storage that stores guest instruction from a guest instruction set, wherein said series includes a guest access instruction that indicates a guest logical address in guest address space;

semantic routine storage that stores a plurality of semantic routines of native instructions for emulating said series of guest instructions;

means, responsive to receipt of said guest memory access instruction for emulation, for translating said guest logical address into a guest real address and for thereafter translating said guest real address into a native physical address; and

means for executing a semantic routine that emulates said guest memory access instruction utilizing said native physical address.

**10**. The processor of claim **9**, wherein said guest memory access instruction comprises one of a guest load instruction and a guest store instruction.

**11**. The processor of claim **9**, wherein said guest memory access instruction comprises a guest instruction that initiates fetching of a guest instruction in said series from said associated memory.

**12**. The processor of claim **9**, wherein said means for executing a semantic routine that emulates said guest memory access instruction comprises means for accessing said memory utilizing said native physical address.

**13**. The processor of claim **9**, said means for translating said guest real address into a native physical address includes means for translating said guest real address into a native effective address and for then translating said native effective address into said native physical address.

**14**. The processor of claim **9**, wherein:

said processor further comprising a translation lookaside buffer (TLB) containing entries utilized for address translation; and

said means for translating said guest logical address into a guest real address and for thereafter translating said guest read address into a native physical address includes:

means for determining if said translation lookaside buffer includes an entry that can be utilized to obtain said native physical address; and

means, responsive to a determination that said translation lookaside buffer contains an entry that can be utilized to obtain said native physical address, for translating said guest logical address into a guest real address and for thereafter translating said guest read address into a native physical address utilizing said translation lookaside buffer (TLB) entry.

**15**. The processor of claim **14**, and further comprising:

means, responsive to a determination that said translation lookaside buffer does not contain an entry that can be utilized to obtain said native physical address, for creating an entry that can be utilized to obtain said native physical address in said translation lookaside buffer.

**16**. The processor of claim **9**, wherein said means for translating includes means for executing a user-level semantic routine.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. :5,953,520
DATED :September 14, 1999
INVENTOR(S) :Soummya Mallick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Col.17, Line 10 please insert after "guest" **--memory--.**

Signed and Sealed this

Fifteenth Day of August, 2000

*Attest:*

*Attesting Officer*

Q. TODD DICKINSON

*Director of Patents and Trademarks*

US005987495A

# United States Patent [19]

Ault et al.

[11] Patent Number: 5,987,495

[45] Date of Patent: Nov. 16, 1999

[54] **METHOD AND APPARATUS FOR FULLY RESTORING A PROGRAM CONTEXT FOLLOWING AN INTERRUPT**

[75] Inventors: **Donald F. Ault**, Hyde Park; **Kenneth E. Plambeck**; **Casper A. Scalzi**, both of Poughkeepsie, all of N.Y.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **08/966,374**

[22] Filed: **Nov. 7, 1997**

[51] **Int. Cl.**[6] ..... **G06F 9/46**

[52] **U.S. Cl.** ..... **709/108**; 710/260

[58] **Field of Search** ..... 709/108, 300; 710/260, 261, 267; 712/208, 233; 711/200

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,912,628 | 3/1990 | Briggs | 364/200 |
| 5,155,853 | 10/1992 | Mitsuhira et al. | 395/734 |
| 5,161,226 | 11/1992 | Wainer | 395/650 |
| 5,375,230 | 12/1994 | Fujimori | 395/575 |
| 5,390,329 | 2/1995 | Gaertner et al. | 395/650 |
| 5,390,332 | 2/1995 | Golson | 395/725 |
| 5,428,779 | 6/1995 | Allegrucci et al. | 395/650 |
| 5,515,538 | 5/1996 | Kleiman | 395/733 |
| 5,761,492 | 6/1998 | Fernando et al. | 395/591 |
| 5,790,872 | 8/1998 | Nozue et al. | 395/740 |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| WO9608948 | 3/1996 | European Pat. Off. . |

OTHER PUBLICATIONS

IBM Technical Disclosure Bulletin, vol. 32, No. 6B—Nov. 1989 "Deterministic Context Switching of Registers" pp. 70–73.

IBM Technical Disclosure Bulletin, vol. 33, No. 3B, Aug. 1990 "Technique To Improve Context . . . In a CPU", pp. 472–473.

Enterprise Systems Architecture/390 Manual—Principles of Operation SA22–7201–02.

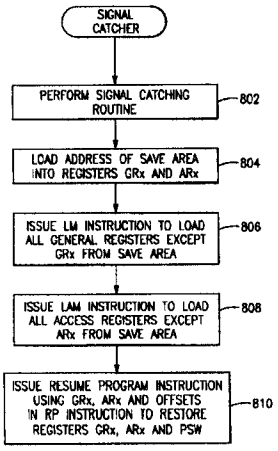
*Primary Examiner*—Gopal C. Ray

*Attorney, Agent, or Firm*—William A. Kinnaman, Jr.

[57] **ABSTRACT**

A method and apparatus for fully restoring the context of a user program, including program status word (PSW) and CPU register contents, following an asynchronous interrupt. Upon the occurrence of an asynchronous interrupt event, control is transferred from the normally executing part of the user program to an interrupt handler of the operating system kernel. The kernel interrupt handler saves the contents of the CPU registers and PSW as they existed at the time of the interrupt in a save area associated with the user program before transferring control to a signal catcher routine of the user program. When it has finished handling the interrupt, the signal catcher routine restores the previous state of program execution as it existed before the interrupt by storing the address of the save area in a selected register (which may be a general register/access register pair), restoring the contents of the registers other than the selected register containing the address of the save area, and then restoring the contents of the PSW and selected register by using a new Resume Program (RP) instruction. The RP instruction contains an operand field specifying through the selected register the base address of the save area together with offset fields specifying the offsets of the saved contents of the PSW and selected register relative to the beginning of the save area. Upon decoding an RP instruction, the CPU executing the instruction adds the displacement to the base address contained in the specified register to form the beginning address of the save area, to which it adds the specified offsets to access the saved PSW and selected register contents. The current PSW and selected register contents are then restored with the saved contents to fully restore the previous program context and return control to the instruction being executed at the point of interrupt. To ensure system integrity, only those fields of the PSW are restored that could have otherwise been restored by a program operating in problem state.

**23 Claims, 6 Drawing Sheets**



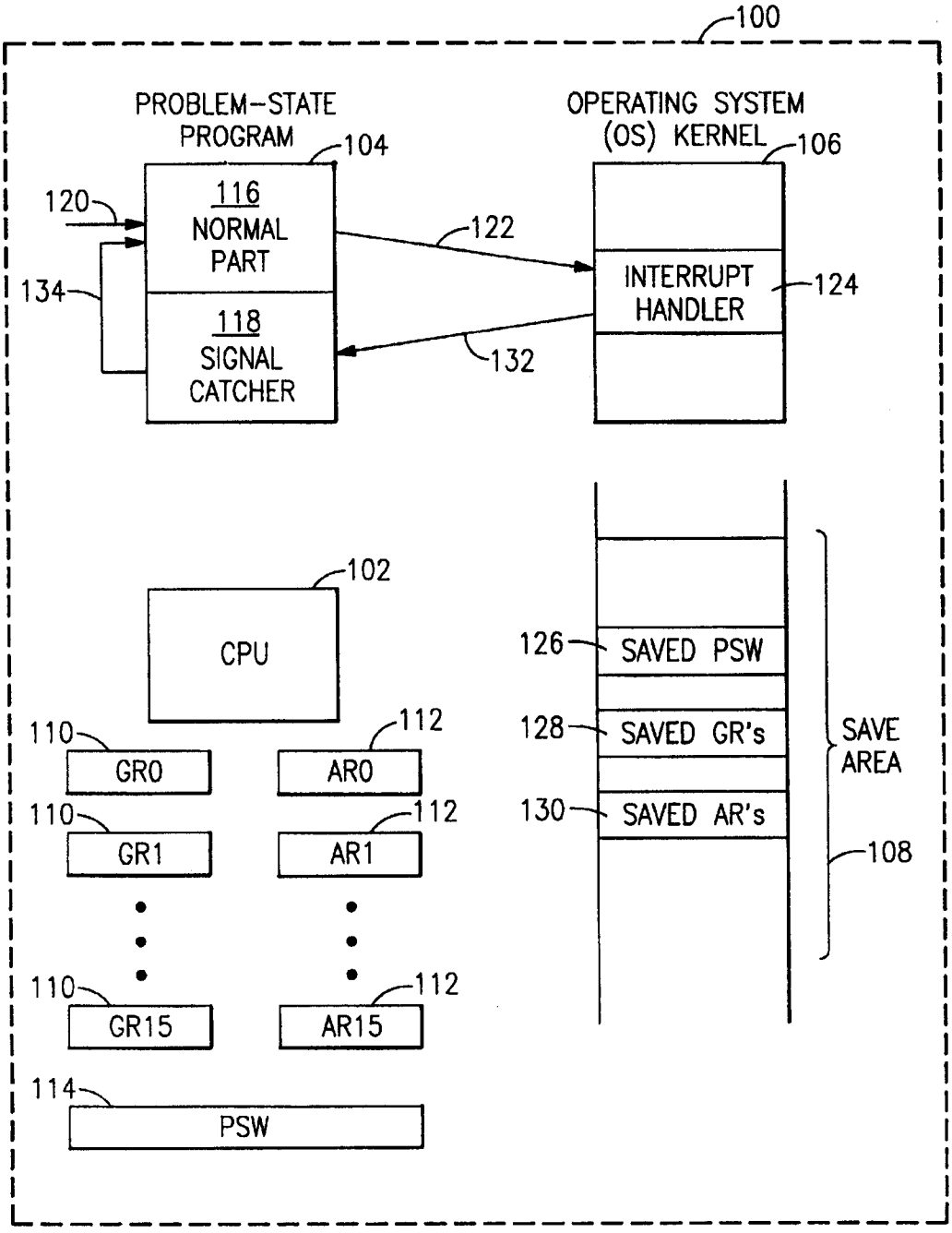

100
PROBLEM-STATE PROGRAM
104
OPERATING SYSTEM (OS) KERNEL
106
120
116 NORMAL PART
122
INTERRUPT HANDLER
124
134
118 SIGNAL CATCHER
132
102
CPU
126
SAVED PSW
128
SAVED GR's
SAVE AREA
130
SAVED AR's
108
110
GR0
112
AR0
110
GR1
112
AR1
110
GR15
112
AR15
114
PSW

FIG.1

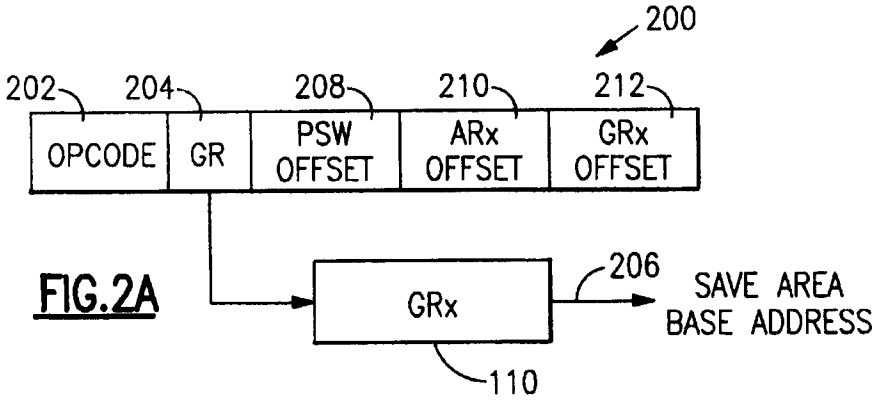

200
202
204
208
210
212
OPCODE
GR
PSW OFFSET
ARx OFFSET
GRx OFFSET
GRx
206
SAVE AREA BASE ADDRESS
110

FIG.2A

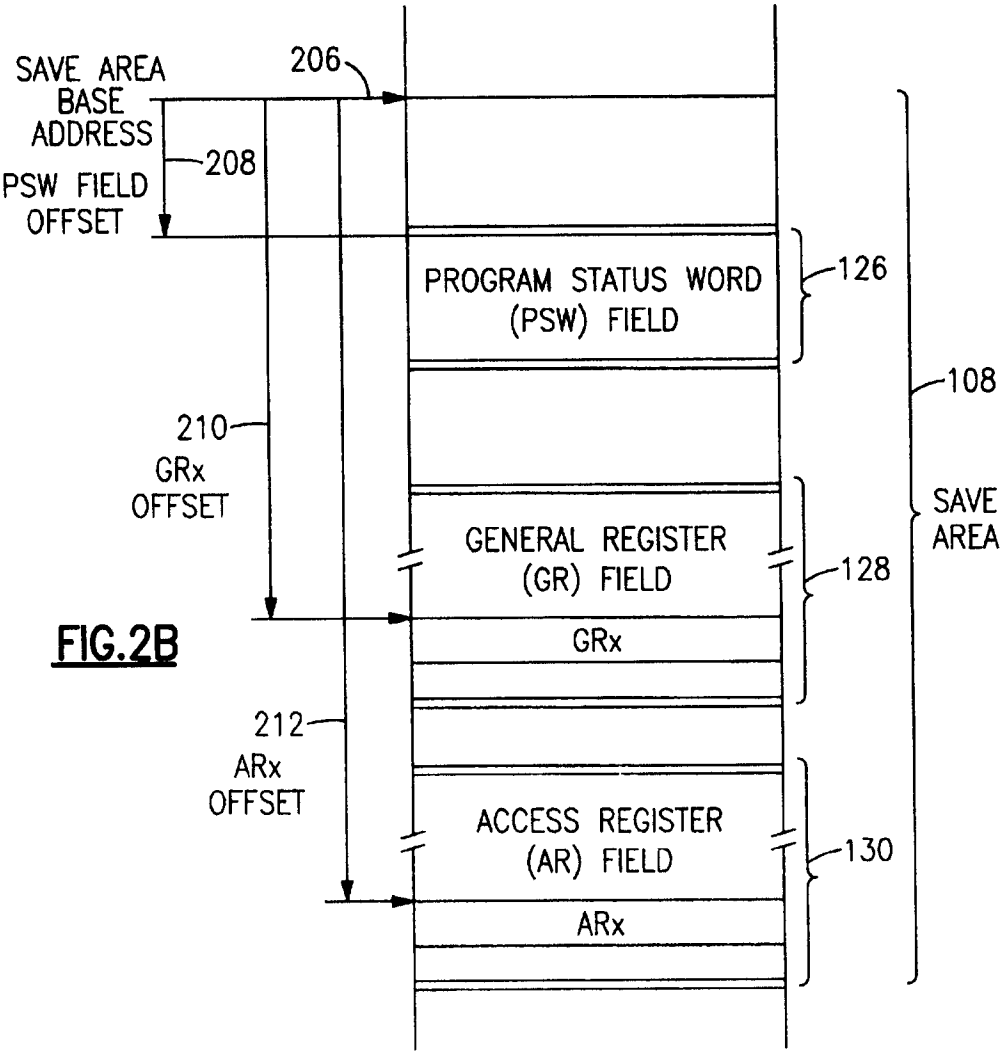

SAVE AREA BASE ADDRESS
206
PSW FIELD OFFSET
208
PROGRAM STATUS WORD (PSW) FIELD
126
108
210
GRx OFFSET
GENERAL REGISTER (GR) FIELD
128
SAVE AREA
GRx
212
ARx OFFSET
ACCESS REGISTER (AR) FIELD
130
ARx

FIG.2B

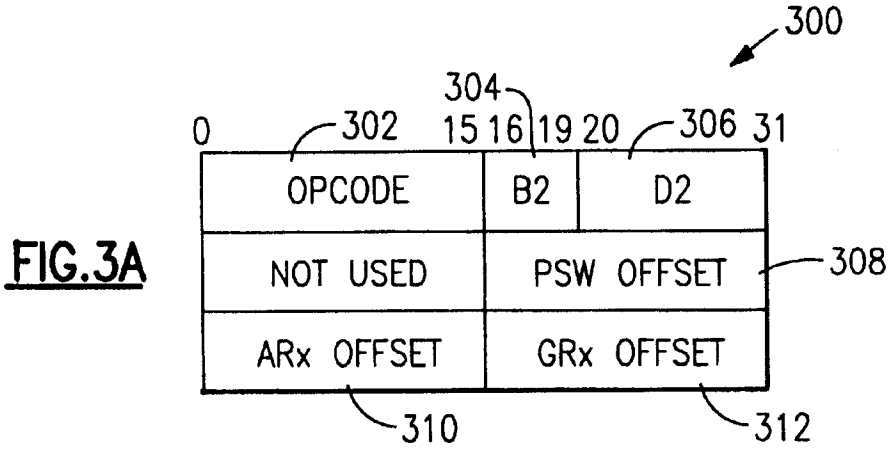

300
304
0
302
15 16 19 20
306 31
OPCODE
B2
D2
NOT USED
PSW OFFSET
308
ARx OFFSET
GRx OFFSET
310
312

FIG.3A

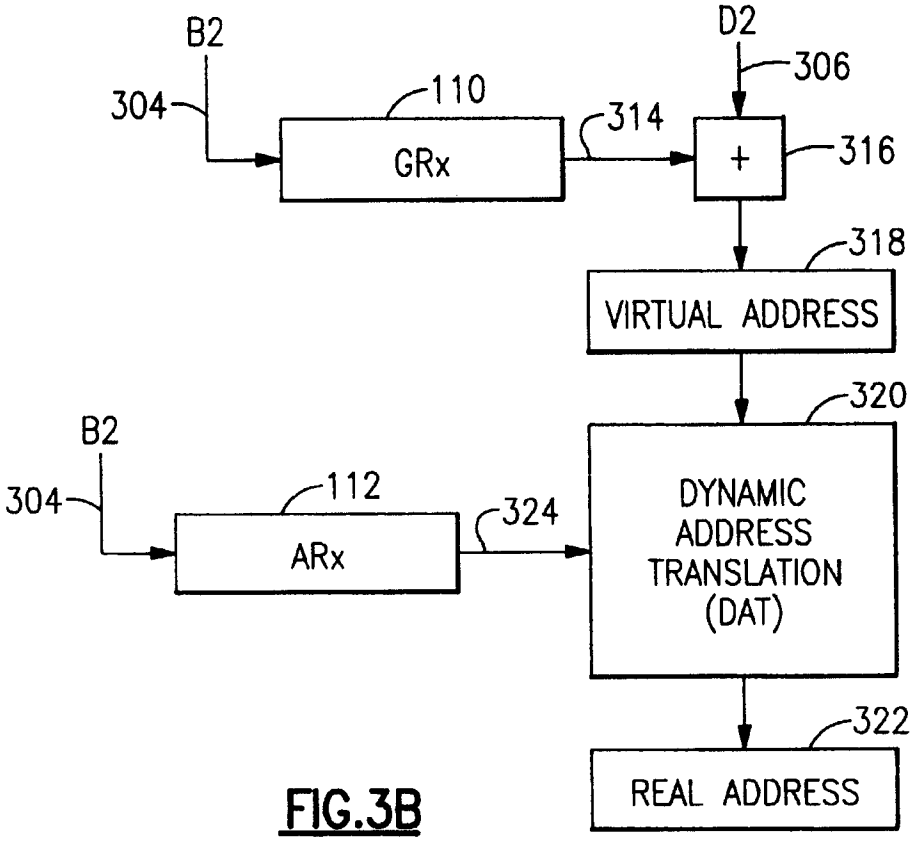

B2
304
110
GRx
314
D2
306
+
316
318
VIRTUAL ADDRESS
320
B2
304
112
ARx
324
DYNAMIC ADDRESS TRANSLATION (DAT)
322
REAL ADDRESS

FIG.3B

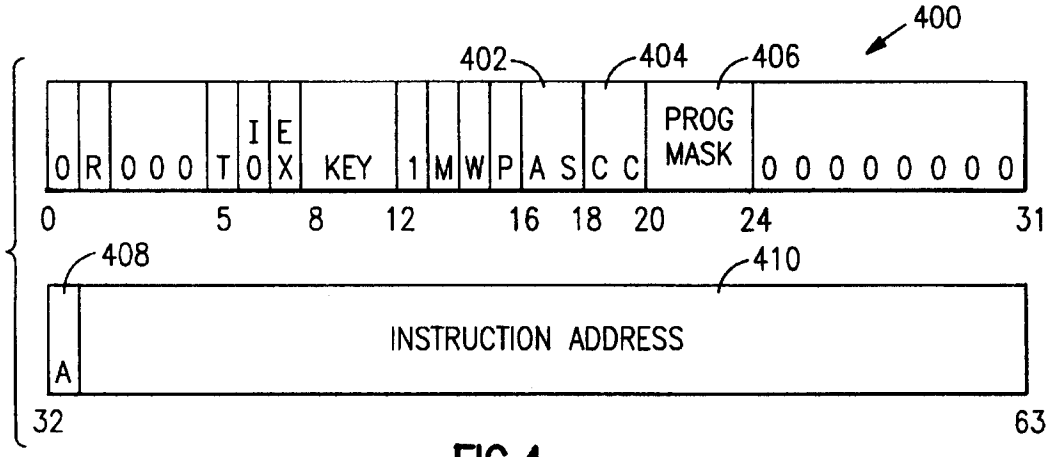



FIG.4

FIG.5

| PSW BITS | FIELD NAME |
|---|---|
| 16 AND 17 | ADDRESS-SPACE CONTROL (AS) |
| 18 AND 19 | CONDITION CODE (CC) |
| 20-23 | PROGRAM MASK |
| 32 | ADDRESSING MODE (A) |
| 33-63 | INSTRUCTION ADDRESS |

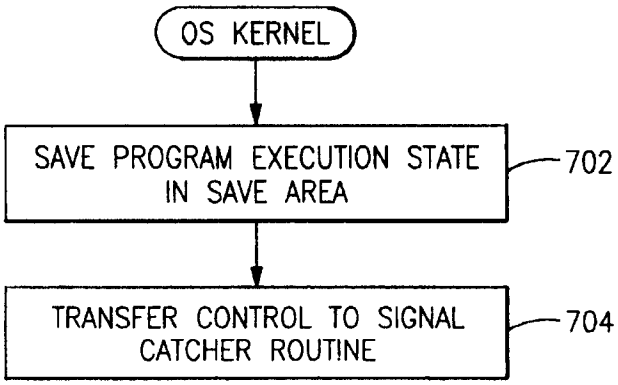



FIG.7

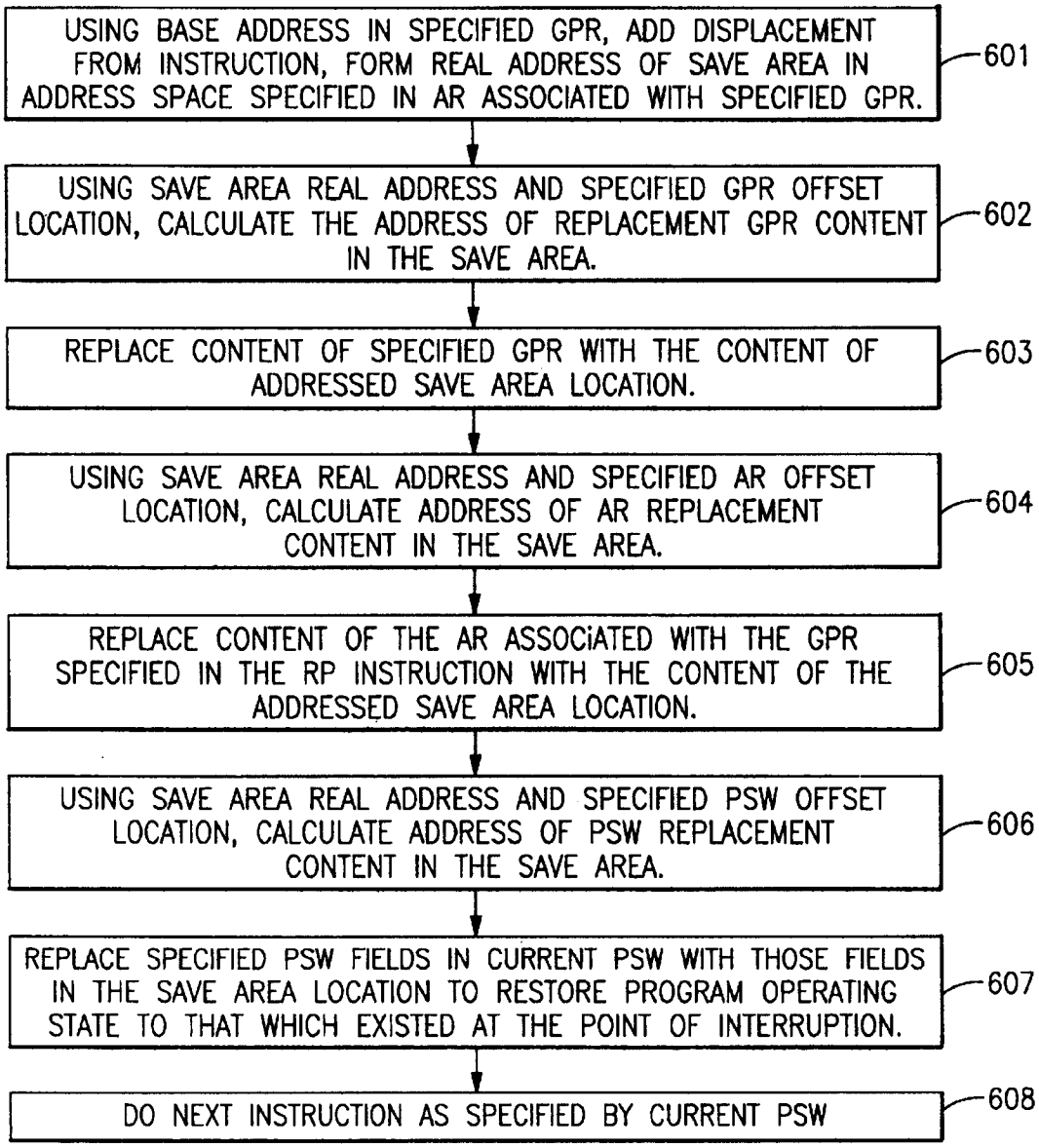

USING BASE ADDRESS IN SPECIFIED GPR, ADD DISPLACEMENT FROM INSTRUCTION, FORM REAL ADDRESS OF SAVE AREA IN ADDRESS SPACE SPECIFIED IN AR ASSOCIATED WITH SPECIFIED GPR.
601
USING SAVE AREA REAL ADDRESS AND SPECIFIED GPR OFFSET LOCATION, CALCULATE THE ADDRESS OF REPLACEMENT GPR CONTENT IN THE SAVE AREA.
602
REPLACE CONTENT OF SPECIFIED GPR WITH THE CONTENT OF ADDRESSED SAVE AREA LOCATION.
603
USING SAVE AREA REAL ADDRESS AND SPECIFIED AR OFFSET LOCATION, CALCULATE ADDRESS OF AR REPLACEMENT CONTENT IN THE SAVE AREA.
604
REPLACE CONTENT OF THE AR ASSOCIATED WITH THE GPR SPECIFIED IN THE RP INSTRUCTION WITH THE CONTENT OF THE ADDRESSED SAVE AREA LOCATION.
605
USING SAVE AREA REAL ADDRESS AND SPECIFIED PSW OFFSET LOCATION, CALCULATE ADDRESS OF PSW REPLACEMENT CONTENT IN THE SAVE AREA.
606
REPLACE SPECIFIED PSW FIELDS IN CURRENT PSW WITH THOSE FIELDS IN THE SAVE AREA LOCATION TO RESTORE PROGRAM OPERATING STATE TO THAT WHICH EXISTED AT THE POINT OF INTERRUPTION.
607
DO NEXT INSTRUCTION AS SPECIFIED BY CURRENT PSW
608

FIG.6

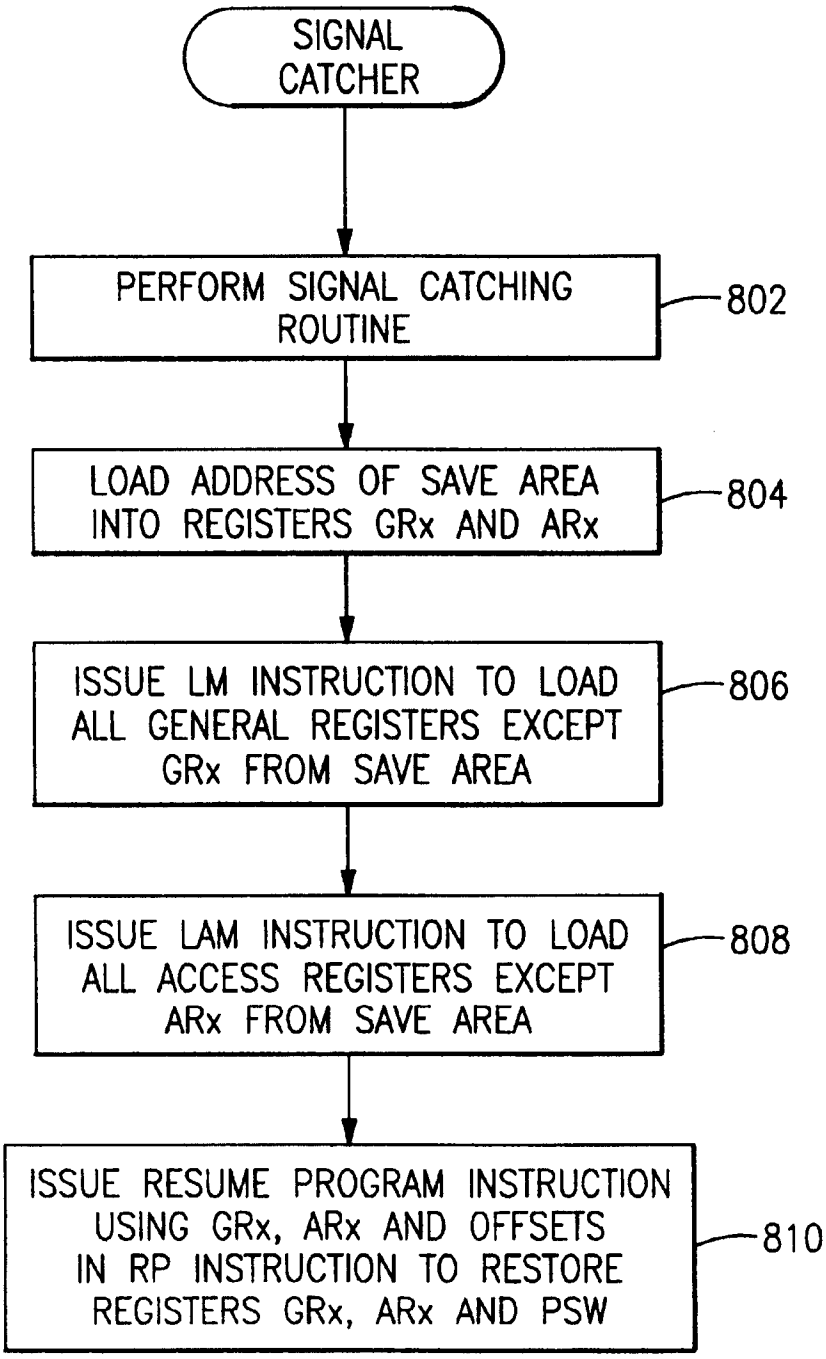

SIGNAL CATCHER
PERFORM SIGNAL CATCHING ROUTINE
802
LOAD ADDRESS OF SAVE AREA INTO REGISTERS GRx AND ARx
804
ISSUE LM INSTRUCTION TO LOAD ALL GENERAL REGISTERS EXCEPT GRx FROM SAVE AREA
806
ISSUE LAM INSTRUCTION TO LOAD ALL ACCESS REGISTERS EXCEPT ARx FROM SAVE AREA
808
ISSUE RESUME PROGRAM INSTRUCTION USING GRx, ARx AND OFFSETS IN RP INSTRUCTION TO RESTORE REGISTERS GRx, ARx AND PSW
810

FIG.8

# METHOD AND APPARATUS FOR FULLY RESTORING A PROGRAM CONTEXT FOLLOWING AN INTERRUPT

## BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to a method and apparatus for fully restoring a program context following an interrupt and, more particularly, to a method and apparatus for restoring the contents of the CPU registers and program status word (PSW) as they existed prior to the asynchronous interrupt of a user program executing in problem state.

2. Description of the Related Art

In a modern computing environment, an operating system is a program (or set of programs) that manages the facilities of a computing system such that the system can be shared among many disparate users being served by multiple independent programs running under the control of that operating system. Hardware resources are under the control of the operating system, which allocates these to the various programs under its control as they request an allocation of them. Thus, real storage space, virtual addressing capability, auxiliary storage space, and ports to the outside world are shared by the programs under the auspices and control of the operating system.

To enforce its management and control over system resources and to provide sharing of facilities with system integrity maintained, the architecture of a system provides mechanisms useful for fencing the operating system from the programs it serves and controls and for fencing those programs from each other. One of these is the operating authority state. Generally, at least two operating states are provided, often called supervisor state and problem state. The supervisor state allows system-wide access authority without fencing, and allows the operating system which uses it to allocate and control which programs have access to which parts of which facilities at which time.

The problem state provides the logical and arithmetic capabilities necessary to solve the problems of a broad range of application programs, and to allow middleware, e.g., database managers or communication access methods, to provide the services to other programs as expected of such middleware. But, in problem state a program is restricted in its accessing capability to that fenced for it by the operating system using the access control mechanisms of the system architecture. These mechanisms are designed to prevent unauthorized access to the operating domain of any other program. Except for performance, and for operating system interfaces expressly provided for intercommunication among programs, the separate programs should not be affected by sharing the system with other programs, and should not be aware of the existence of the other programs. Because of the prevalence of programming error, caused by the complexity of some programming, middleware generally operates in problem state for most of its operating time in order to isolate each such program from the others, in order to minimize the effect of the occasional error, ease detection of the cause of such errors, and improve the recoverability of the system when such errors occur. Further, application programs must be authorized only to those system aspects that affect their own execution. This is particularly true in a world in which computer viruses are seen, and in which, though infrequent, other cases of programming malice are experienced.

Although the present invention may be used in other architectures, it will be discussed in the setting of the IBM® S/390® architecture as documented, for example, in the IBM publication *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201-02, 1994, and successor versions thereof, incorporated herein by reference.



One of the key mechanisms in an S/390 system is the program status word (PSW), which directs the processor in the execution of a program. It indicates the next instruction to be executed and contains controls constraining the operating state and authority of the program executing under that PSW. Another mechanism is virtual addressing, where the operating system supplies the real backing storage for the virtual storage accessed by problem state programs. Another control mechanism is the set of control words that determine new PSW content on events that must be handled asynchronously by the operating system. For example, when the processor wishes to present a signal that an input/output (I/O) operation has completed and the device or control unit wishes to make a report of the event, this area will indicate the instruction location of the first instruction of the routine that handles the event. The operating system must handle this external event since the I/O devices as a group are shared with different programs allowed access to different ones. The operating system must reflect the completion, in accordance with its own protocols, to programs requiring notification. The PSW content established on the occurrence of an event to be handled by a part of the operating system generally puts the system into the supervisor state, but the interrupted state is saved for later reestablishment when the interrupted program is later to be resumed. Most of the time, the interrupted program was one executing in problem state and restoring state will return the processor to that state. Because the PSW is used to constrain the capability of the program executing on a processor, loading the PSW is restricted to programs executing in supervisor state. One obvious reason is that, depending on the setting of its problem state bit, the PSW authorizes supervisor state or restricts the executing program to problem state with its access and operational restrictions. The restrictions imposed on a problem state program would be of little consequence if the program could simply upgrade its state to supervisor state by overwriting the problem state bit in the PSW.

There are sound technical reasons for allowing a complex program, running in a system with an operating system, to itself contain asynchronous processes associated with asynchronous events for which the program provides special event processing, but to execute in problem program authority state nonetheless, for system integrity reasons. One example occurs in UNIX® programs in which one program may send a message or signal to another program, with the signal arrival occurring asynchronously to the normal processing of the program which is to receive the signal. The kernel program interrupts the normal flow of the program to which it is to deliver the signal and transfers control to a different part of the program designed and coded to handle the asynchronous arrival of the signal. We can call this part of the program its signal catcher routine. The problem posed is that of an efficient return to the normal operating part of the program at its point of interruption after the signal handling part of the program has completed its processing of the signal event. When the operating system kernel handles a logical interruption to an executing program, it has saved the operating state of that program, allowing later resumption of the program, as if the interruption never occurred. In an S/390 system, this involves saving all general purpose resisters (GRs), access registers (ARs), the content of the PSW at the point of interruption, including both the instruction address of the point of interruption and the state

variables controlling the execution of the program. The PSW also records the current setting of the condition code, which reflects the kind of result obtained in the last arithmetic or logical operation, or special circumstances arising in other types of instruction. The program mask, indicating how the processor should behave when certain program exceptions occur during the performance of certain instruction types is also part of the PSW, and actions by the program, which do not require any special authority, can change bits in this field. These are set by the program in concert with its own structure, and each program may have a different program mask and may change it from time to time without communication with the operating system, in order to change the handling of an exception condition. The PSW also specifies the addressing mode, i.e., whether the processor should produce 24-bit addresses or 31-bit addresses when forming effective addresses. This can be changed freely as part of certain branch instructions, so the mode may be either value at any time, and must be restored to that value after an interruption if the program is to operate correctly. The PSW also indicates whether a problem state program is in primary space mode or access register mode at any point in its execution, and this must be properly restored if the program is to execute correctly. Since a problem state instruction can be used to switch between the two addressing modes, the program may be in either mode at any time, unpredictably, and after an interruption, the correct value must be restored.

In the S/390 operational environment, the UNIX kernel itself operates as part of the operating system, and has saved the status of the interrupted program at the point of its interruption. The save area contains the PSW contents as well as the general registers (GRs) and access registers (ARs). Since this save area is provided to allow what is essentially an emulation of an interruption within a single problem state program, it will be preserved should the signal handling part of the program be itself interrupted after it has been entered to handle the signal received. The operating system will use another save area should an interruption occur while the signal handling routine is executing. The save area to be used in returning from the signal handling part of the program to its interrupted part is preserved in storage for that process, in an area accessible by the program itself with its normal storage access authority.

In an S/390 system, which uses the general registers for specifying the addresses of storage operands, it is impossible for a problem state program to transfer control directly to another program using a normal branch instruction and, at the same time, restore all the general registers to some saved earlier value. That is because the save area address is specified by the contents of a general register/access register (GR/AR) register pair, and these values were not the content of the registers at the earlier time of saving the register contents, in the most general case. Also, the branch address must be specified in another general register whose content would generally have been different at the time of saving the registers. Also, it is impossible to properly reflect the control fields of the PSW as they were at the time of the interruption without the use of a Load PSW instruction, which requires the issuing program to be in supervisor state. This is particularly true of the condition code field in the PSW.

The problem has been solved within the operating system since it must perform such actions routinely in dispatching programs. It does this by the use of a PSW in low storage which can be accessed without use of a GR, after disabling the system's interruption capability so that it can not be interrupted in the middle of restoring the execution state of an interrupted program. It would be possible for an operating system service to be defined that would perform the restoration of control back to the interrupted part of the program that the signal catcher is part of, but this would require transition from problem state to supervisor state, and establishment of the PSW to be restored in the low storage area of the computing system, and use of the Load PSW instruction, with the performance negatives of such an instruction path.

What is desired instead is a processor mechanism that provides a direct resumption of an earlier interrupted program without disabling the processor from hardware interruption handling, and without requiring the program to be in an authorized state to cause the resumption from the logical interruption, and without causing a transition to an authorized state to have it done by an authorized system service. It is estimated that such a mechanism would save hundreds, and perhaps even thousands, of executed instructions in doing the program control restoration to the program at the point at which it was logically interrupted for the signal delivery.

## SUMMARY OF THE INVENTION

As a preliminary to describing the invention itself, the insights that led to it will be discussed. Problem state programs generally have the least authority among the agencies of the computing system. Only a subset of the architected facilities are available to it. The defined architecture of the system provides the operating system with a set of architected facilities which it may use, and in some cases allocate to problem state programs. Some of those that are withheld from direct use are made available by means of system services provided by the operating system, particularly where physical resources are shared by different programs, e.g., real main storage, external storage space, networking facilities, etc. In like manner, at a lower level of control, the microcode and hardware agencies of the system have defined facilities for their own use which are not accessible, or even seen at the system architecture level. Examples are a section of storage not accessible to programs in either problem or supervisor state, which storage is required to perform the invocable functions of the architecture, registers reserved for internal use, adders and other logical units needed to do addressing and arithmetic. The microcode in particular must perform its assigned programming tasks without polluting the architected facilities of the system. It operates at a third, separate and isolated, level of control in the system, with capabilities beyond the problem state and the supervisor state. The microcode and hardware of the system are designed and the design verified and tested for correctness before the system is manufactured. Special concern is paid that the hardware and microcode can not be compromised as far as system integrity is concerned by actions performed by programs in either supervisor or problem state. Otherwise, the authority structure of the system architecture can not be guaranteed.

The major point here is that the microcode does not change dynamically, it can be considered to be a manufactured component of the system, and is a fully trusted element in the integrity structure of the system. The microcode has full access to programmable storage, and to the Program Status Word (PSW) of the system as part of its necessary capabilities. The microcode can perform complex operations of many steps in order to provide the functions of what is a single instruction at the architecture level above it. Similarly, below the microcode level of control, the hardware elements of the system must have complete access to all system facilities, even those not readily available to the microcode.

The constraint on them is that they may not allow a higher agency to use the facilities usable at the higher level to compromise the integrity of their own operation.

Therefore, in accordance with the present invention, a new instruction, referred to herein as Resume Program (RP), which is invocable in problem state, and is performed at either the microcode or the hardware level, is defined to perform the transition of control from the interrupt-handling routine back to the point of interruption in the main path of the program by restoring state information saved in a save area. The instruction is defined such that only aspects of the PSW that are changeable by a problem state program by other means can be restored using it. Also, since the condition codes of the programming level PSW are not set by the microcode or hardware levels for their own purposes, and the PSW of the programming level is not used by them, they have no problem in establishing a restored state in the PSW.

The instruction addresses the save area by means of a register, and in S/390, possibly an access register (AR), which general register and access register are to be restored by the RP instruction, after using the save area address they specify for the RP instruction itself. The instruction specifies the offsets within the save area of the saved PSW content, the AR to be restored, and the general register to be restored. Only the specified PSW fields (those changeable within problem state) are restored from the saved PSW to the PSW that is given control when the instruction completes its own execution.

More particularly, in one aspect the present invention contemplates a method and apparatus for operating a processor to restore a previously saved program context in an information handling system in which execution of a program by the processor is controlled by a program status word (PSW) defining a program context, in which the program executes in either a first state having relatively restricted authority or a second state having relatively unrestricted authority, and in which the PSW contains a first set of fields that are alterable by a program executing in the first state and a second set of fields that are not alterable by a program executing in the first state. In accordance with this aspect of the invention, a Resume Program (RP) instruction is defined that specifies a storage location containing a saved PSW. Upon decoding an RP instruction, the processor restores from the saved PSW word contained at the specified storage location only those fields of the current PSW that are alterable by a program executing in the first state.

The saved PSW may contain an instruction address that is restored to cause execution to resume at that address. The RP instruction is intended for execution by a program executing in the first state having relatively restricted authority, hence the restrictions on which fields of the PSW are updated. Preferably, the RP instruction contains a field specifying a register, and the storage location is determined using the contents of the specified register. The field may be a first field, and the RP instruction may contain a second field specifying a displacement from a base address, in which case the storage location is determined by adding the displacement contained in the second field to a base address contained in the register specified by the first field.

The save area may also contain the saved contents of the register itself, which are restored to the register from the save area. The RP instruction may specify a beginning address of the save area and an offset from the beginning address, with the storage location being determined by adding the offset specified by the RP instruction to the beginning address of the save area.

Another aspect of the present invention contemplates a method and apparatus for operating a processor to restore a previously saved program context comprising a PSW and a set of register contents. In accordance with this aspect of the invention, the Resume Program (RP) instruction specifies a register selected from the set of registers that points to a save area containing a saved PSW and saved register contents. In an S/390 environment or other environment using access registers in a similar manner, the specified register may be a general register/access register (GR-AR) pair. In response to decoding an RP instruction, the processor accesses the save area using the contents of the specified register and then restores the PSW and the register from the saved PSW and saved register contents contained in the save area. This causes the processor to resume execution at the instruction address contained in the saved PSW with the program context defined by the saved PSW and saved register contents.

As before, the specified register may specify a base address, and the RP instruction may also specify a displacement that is added to the base address to obtain the address of the save area. Likewise, the RP instruction may specify offsets that are added to the beginning address of the save area to obtain the addresses at which the various saved values (PSW, register contents) are stored.

Yet another aspect of the invention contemplates a method whereby a program may use the RP instruction to restore a previously saved PSW and register contents defining a previous program context. In accordance with this aspect of the invention, the program loads the address of the save area containing said previously saved program context into a specified register (which may be a GR/AR pair), restores the contents of the registers of each register type other than the specified register from the save area using a first instruction (e.g., LM for general registers and LAM for access registers in an S/390 environment), and restores the contents of the PSW and the specified register from the save area using the RP instruction to resume execution at the instruction address contained in the saved PSW with the program context defined by the saved PSW and saved register contents.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows a computer system in which the present invention may be used.

FIG. **2**A depicts the format of a general Resume Program instruction of this invention.

FIG. **2**B shows the address arithmetic involved in executing the instruction of FIG. **2**A.

FIG. **3**A depicts a Resume Program instruction for an S/390 embodiment described herein.

FIG. **3**B shows the address arithmetic involved in executing the instruction of FIG. **3**A.

FIG. **4** depicts an S/390 Program Status Word (PSW) used in the described embodiment.

FIG. **5** lists the PSW fields that are restored by the execution of the Resume Program (RP) instruction in an S/390 embodiment.

FIG. **6** is a flow chart of the execution of a Resume Program (RP) instruction.

FIG. **7** shows the steps performed by the interrupt handler of the OS kernel.

FIG. **8** shows the steps performed by the signal catcher routine that invokes the Resume Program (RP) Instruction.

## DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. **1** shows a computer system **100** in which the present invention may be used. As shown in the figure, computer

system **100** contains a central processing unit (CPU) **102**, at least one user program **104**, an operating system (OS) kernel **106**, and a save area **108** associated with the user program **104**, which has the necessary authorization to access it. Computer system **100** may comprise either a separate physical machine or a separate partition of a logically partitioned machine. Although the invention is not limited to any particular hardware platform, it will be discussed in the exemplary context of an IBM S/390 environment. In such an environment, system **100** may be an IBM S/390 Parallel Enterprise Server, while OS kernel **106** may comprise the IBM OS/390 operating system.

In an S/390 environment, it will be assumed that user program **104** is executing in what is known as the problem state (and is hence referred to as a "problem state program"), while OS kernel **106** operates normally in what is known as the supervisor state. As explained in the S/390 architecture document referred to above, in the supervisor state all instructions are valid, whereas in the problem state only unprivileged instructions and (if certain authority tests are met) semiprivileged instructions are valid.

CPU **102**, which constitutes the primary instruction processing unit of system **100**, may comprise one or more central processors (CPs) (not separately shown). As is conventional in the art, CPU **102** has an instruction decoder for decoding instructions being executed as well as an execution unit for executing the decoded instructions. These may be implemented by any suitable combination of hardware and microcode in a manner well know in the art. Since the details of their construction and operation form no part of the present invention, they are not separately shown. CPU **102** has an instruction set that (except for the present invention) is generally described in the architecture document referred to above. This instruction set, or architecture, defines how the CPU **102** appears to programming such as user program **104** or OS kernel **106**. As noted in the summary portion above, the hardware and microcode implementing the processor architecture, because of their relative immutability, constitute highly "trusted" parts of the system **100**, as contrasted with the OS kernel **106** (which is accorded an intermediate level of "trust") or user program **104** (which is accorded the lowest level of "trust").

Associated with CPU **102** are a set of 16 32-bit general registers **110** (GR0–GR15), 16 32-bit access registers **112** (AR0–AR15), and a 64-bit program status word (PSW) **114**. General registers **110** are used as base address registers and index registers in address arithmetic and as accumulators in general arithmetic and logical operations. Access registers **112** are used to specify segment table designations used to perform dynamic address translation. PSW **114** stores the address of the next instruction to be executed, along with other pertinent state information, such as a condition code and various settable program modes, as described below. In addition to registers **110** and **112** and PSW **114**, CPU **102** has other registers (such as control registers and floating-point registers) that are not relevant to the present invention and are hence not shown.

Problem state program **104** contains a first part **116** that is executed normally and a second part **118** (referred to as a "signal catcher" herein) that is executed in response to an interrupt. More particularly, in response to an asynchronous event at **120** (such as a message or signal from another program), control is transferred at **122** from the problem state program **104** to an interrupt handler **124** of the OS kernel **106** executing in supervisor state. The interruption point at **120** may be arbitrary with respect to the contents of registers **110** and **112** and PSW **114**, which cannot be assumed to be any particular value.

In the preferred embodiment, interrupt handler **124** may be a UNIX signal-handling kernel program. Referring also to FIG. **7**, upon gaining control, interrupt handler **124** saves the contents of the general registers **110**, access registers **112** and portions of PSW **114** (together constituting what will be referred to as the program execution state or program context) as they existed at the point of interrupt in save area **108** (step **702**). More particularly, the saved portions of PSW **114** are saved in a saved PSW location **126** of the save area **108**, the contents of general registers **110** are saved in a saved GR location **128** of the save area, and the contents of access registers **112** are saved in a saved AR location **130** of the save area. The interrupt handler **124** then transfers control at **132** to the signal catcher routine **118** in the problem state program **104** (step **704**).

Upon gaining control, the signal catcher **118** first makes a copy of the passed in save area **108**, and then enables recursive signals. That way, each instance of the signal catcher **118** has its own resume save area **108**. Thus, signal catcher **118** can be interrupted by a signal, which can be interrupted by another signal, and so on, in a recursive manner as described above.

After this initialization, signal catcher **118** performs its functions (the particulars of which form no part of the present invention) for processing the event that occurred at **120**. When signal catcher **118** completes its processing of the interrupt, one of its options is to return at **134** to the point of interruption at **120** to continue normal processing as if an interruption had not occurred.

FIG. **8** shows the steps performed by the signal catcher **118** to return control to the normal part **116** of the program **104** at the point of interruption **120** after the signal catcher has performed its function. Referring to the figure, the signal catcher **118** first loads the address of the save area **108** into a selected general register/access register pair GRx/ARx, where x is an index ranging between 1 and 15 in the embodiment shown (step **802**). (In an S/390 environment, x cannot be 0 because GR0 cannot be used for addressing.) Next, the signal catcher **118** restores from the save area **108** all registers **110** and **112** that are not needed by the Resume Program (RP) instruction **300** itself to access the save area. In an S/390 environment, this is done by issuing a Load Multiple (LM) instruction to restore the contents of the set of general registers GRi (where $i \neq x$) from the GR field **128** of the save area **108** (step **804**), as well as issuing a Load Access Multiple (LAM) instruction to restore the contents of the set of access registers GRi (where $i \neq x$) from the AR field **130** of the save area **108** (step **806**). Although step **804** is shown as preceding step **806** in FIG. **8**, the particular order in which the steps are performed is immaterial.

At this point the signal catcher **118** invokes the Resume Program (RP) instruction **300** of the present invention to do full context restoration to the point of the interruption (step **808**). The RP instruction **300** uses the content of the general register GRx (and access register ARx, if needed) specified in the instruction to access the save area **108**, in the manner described in detail below.

FIG. **2A** shows the logical format **200** of the Resume Program (RP) instruction of the present invention, which is not specific to any particular platform. In the logical format **200**, an operation code (OPCODE) **202** identifies the instruction as an RP instruction; a register specification (GR) **204** specifies a general register **110** (GRx) that contains the base address **206** of the save area **108**; a PSW offset **208** specifies the offset of the saved PSW contents **126** from the beginning of the save area **108**; an ARx offset **210** specifies

the offset of the saved ARx contents from the beginning of the save area **108**; and a GRx offset **212** specifies the offset of the saved GRx contents from the beginning of the save area **108**.

FIG. **2**B shows graphically the address arithmetic involved in using the operands **204**–**212**. As shown in the figure, to generate the beginning address of the PSW field **126**, the PSW offset **208** is added to the base address **206** contained in the general register **110** (GRx) pointed to by the GR field **204**. Similarly, to generate the address of the saved GRx contents in field **128**, the GRx offset **210** is added to the base address **206** contained in the general register GRx. Finally, to generate the address of the saved ARx contents in field **130**, the ARx offset **212** is added to the base address **206** contained in the general register GRx. Although not shown in FIGS. **2**A and **2**B, the base address **206** contained in general register GRx may be a virtual address that is converted into a real address by dynamic address translation (DAT). In such a case, the corresponding access register ARx may be used to specify a particular address space for which the conversion is performed or to otherwise control the address translation.

FIG. **3**A illustrates a possible instruction format **300** for an S/390 environment, while FIG. **3**B shows the additional address arithmetic implied by the format. Format **300** comprises a 32-bit instruction proper (bits **0**–**31**), followed immediately by a 64-bit parameter list that for practical purposes may be regarded as part of the instruction. The instruction proper contains a 16-bit opcode field **302**, followed by a 4-bit base register field **304** (B2) and a 12-bit displacement field **306** (D2). The 64-bit parameter list contains a 16-bit unused field (filled with zeros), followed by a 16-bit PSW offset **308**, a 16-bit ARx offset **310** and a 16-bit GRx offset **312**.

Fields **302** and **308**–**312** are similar to the corresponding fields **202** and **208**–**212** in logical format **200** and will not be redescribed. Field **304** (B2), like field **204** in logical format **200**, specifies a particular general register **110** (GRx) used to point to the save area **108**. However, the address contained in the specified register GRx, rather than pointing directly to the beginning address of save area **108**, is a base address **314** that is combined at **316** with a displacement D2 specified in field **306** to obtain a beginning address **318** for the save area. As with the address **206**, beginning address **318** may be a virtual address that is converted by dynamic address translation (DAT) **320** to a real address **324**. As suggested above for the logical format **200**, the dynamic address translation **320** may depend on an address space specification determined by an input **324** from the corresponding access register **112** (ARx). The particulars of the dynamic address translation **320** are described in the S/390 architecture document identified above. Except for the fact that the dynamic address translation **320** is determined in part by the contents **324** of the access register ARx, its particulars form no part of the present invention and are hence not discussed in this specification.

As described below, execution of the RP instruction **300** causes certain fields in the current PSW **114** and the contents of the access register **112** (ARx) and general register **110** (GRx) specified by the index in field **304** (B2) to be replaced with fields in the save area **108** (as specified by the second operand address B2, D2) having the specified offsets **308**–**312**.

FIG. **4** shows the format of a conventional S/390 program status word (PSW) **400**, as described, for example, in the architecture document referred to above. PSW **400** contains several fields of interest to the present invention, since they are saved in the PSW portion **126** of save area **108** following an interrupt. These fields include an address space control (AS) **402** (bits **16**–**17**); a condition code (CC) **404** (bits **18**–**19**); a program mask **406** (bits **20**–**23**); an addressing mode (A) **408** (bit **32**); and an instruction address **410** (bits **33**–**63**). The address space control (AS) **402** specifies, in conjunction with fields in the control registers and control blocks, the instruction address space and the address space containing storage operands. The condition code (CC) **404** is set as a result of certain arithmetic operations and comparisons and can be used to do conditional branching so as to direct program flow based on past results. The program mask **406** specifies, for certain arithmetic results, whether or not those results should cause an interruption, either for terminating program execution or for modifying the results. The addressing mode (A) **408** specifies either a 24-bit or a 31-bit addressing mode. The instruction address **410** is the address of the next instruction to be executed.

PSW **400** contains other fields, which are not restored by the Resume Program instruction of the present invention, since they are not alterable by a program executing in problem state. These include bits **0**–**15**, of which bit **15** is the problem state (P) bit defining whether the CPU **102** is in the problem state (P=1) or in the supervisor state (P=0).

FIG. **5** lists the fields of PSW **400** that are restored by Resume Program from the PSW in the save area used by Resume Program in an S/390 embodiment. As indicated above, the restored fields include the address space control **402**, the condition code **404**, the program mask **406**, the addressing mode **408**, and the instruction address **410**.

FIG. **6** is a flowchart of the operation of the Resume Program (RP) instruction **300**. These actions are performed at the microcode and hardware level of the system **100**, within CPU **102**; the particulars of their implementation at these levels form no part of the present invention and are hence not shown. Changes to the architected level can not be seen by the program **104** until the RP instruction **300** has completed and control is returned to the program **104** using the PSW **114** as modified by the RP instruction.

Upon decoding an RP instruction, CPU **102** obtains the real address **322** (FIG. **3**B) of the save area **108** by forming the effective virtual address **318** and performing normal address translation **320** (step **601**).

Next, the address in the save area **108** of the saved value of the general register GRx (specified by the B2 field **304** of the RP instruction) is calculated using the base address **322** of the save area and the GRx offset **312** specified in the instruction parameter list (step **602**). This address is then used to replace the content of the general register GRx with the saved content at the addressed save area location (step **603**).

The procedure of steps **602**–**603** is then repeated for the access register ARx associated with the general register GRx. Using the save area real address **322** and the ARx offset **310** specified in the parameter list, the address in the save area **108** of the saved content of the access register ARx is calculated (step **604**). The content of the access register ARx is then replaced with the content from the addressed save area location (step **605**).

The procedure is then repeated once again to restore the PSW **114**. The real address of the PSW **126** in the save area **108** is calculated by using the save area real address **322** and the PSW offset **308** specified in the RP parameter list (step **606**). Then, the fields specified for change in the RP instruction definition (FIGS. **4**–**5**) are replaced by the correspond-

ing fields in the stored PSW **126** in the save area **108** (step **607**). These fields include the instruction address **410** of the next instruction to be executed in the interrupted part **116** of the program **104**; therefore, the RP instruction **300** in effect causes a branch.

Finally, following execution of the RP instruction **300**, the next instruction in the interrupted program **104** is executed as specified by the restored PSW **114** (step **608**).

As already noted, the RP instruction described above, like other aspects of the CPU architecture, may be implemented by hardware, by microcode, or by any suitable combination of the two, while the signal catcher **118** utilizing the RP instruction is preferable implemented as software. (Both microcode and software constitute programming, the principal difference being that microcode implements an architectural interface while software interacts with it.) While a particular embodiment has been shown and described, those skilled in the art will appreciate that various modifications may be made without departing from the principles of the invention.

What is claimed is:

**1**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word defining a program context, said program executing in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said program status word containing a first set of fields that are alterable by a program executing in said problem state and a second set of fields that are not alterable by a program executing in said problem state, a method of operating said processor to restore a previously saved program context, comprising the steps of:

decoding an instruction from a program executing in said problem state specifying a storage location containing a saved program status word; and

in response to decoding said program instruction, restoring from the saved program status word contained at said specified storage location only those fields of the current program status word that are alterable by a program executing in said problem state.

**2**. The method of claim **1** in which said program instruction contains a field specifying a register, said restoring step comprising the step of:

determining said storage location using the contents of the register specified by said field.

**3**. The method of claim **2** in which said field is a first field, said program instruction containing a second field specifying a displacement from a base address, said restoring step comprising the step of:

determining said storage location by using the contents of the register specified by said first field as a base address and adding to said base address the displacement contained in said second field.

**4**. The method of claim **2** in which said register specifies a save area containing said storage location and saved contents of said register, said restoring step comprising the further step of:

restoring said saved register contents to said register from said save area.

**5**. The method of claim **1** in which said program instruction specifies a beginning address of a save area and an offset from said beginning address, said restoring step comprising the step of:

determining said storage location by adding the offset specified by said program instruction to the beginning address of the save area specified by said program instruction.

**6**. The method of claim **1** in which said saved program status word contains an instruction address that is restored by said restoring step to cause execution to resume at said instruction address.

**7**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, a method of operating said processor to restore a previously saved program context, comprising the steps of:

decoding a program instruction specifying a register selected from said set of registers, said register pointing to a save area containing a saved program status word and saved register contents, and

in response to decoding said program instruction:

accessing said save area using the contents of the register specified by said program instruction; and

restoring said program status word and said register from the saved program status word and saved register contents contained in said save area to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

**8**. The method of claim **7** in which said program instruction specifies a general register.

**9**. The method of claim **7** in which said program instruction specifies a general register and an access register, said accessing step using the contents of said general register and said access register to access said save area.

**10**. The method of claim **7** in which the specified register specifies a base address, said program instruction also specifying a displacement from said base address, said accessing step comprising the step of:

adding the specified displacement to the base address contained in said specified register.

**11**. The method of claim **7** in which said register specifies a beginning address of said save area, said program instruction also specifying an offset from said beginning address, said accessing step comprising the step of:

adding the offset specified by said program instruction to the beginning address of the save area specified by said register.

**12**. The method of claim **7** in which said program executes in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said restoring step being performed for a program executing in said problem state and restoring only those fields of the current program status word that are alterable by a program executing in said problem state.

**13**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, a method of restoring a previously saved program status word and saved register contents defining a previous program context, comprising the steps of:

loading the address of a save area containing said previously saved program context into a specified one of said registers;

restoring the contents of said registers other than said specified register from said save area using a first instruction; and

restoring the contents of said program status word and said specified register from said save area using a

second instruction to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

**14**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word defining a program context, said program executing in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said program status word containing a first set of fields that are alterable by a program executing in said problem state and a second set of fields that are not alterable by a program executing in said problem state, apparatus for operating said processor to restore a previously saved program context, comprising:

means for decoding an instruction from a program executing in said problem state specifying a storage location containing a saved program status word; and

means responsive to said decoding means for restoring from the saved program status word contained at said specified storage location only those fields of the current program status word that are alterable by a program executing in said problem state.

**15**. The apparatus of claim **14** in which said program instruction contains a field specifying a register, said restoring means comprising:

means for determining said storage location using the contents of the register specified by said field.

**16**. The apparatus of claim **15** in which said field is a first field, said program instruction containing a second field specifying a displacement from a base address, said restoring means comprising:

means for determining said storage location by using the contents of the register specified by said first field as a base address and adding to said base address the displacement contained in said second field.

**17**. The apparatus of claim **15** in which said register specifies a save area containing said storage location and saved contents of said register, said restoring means further comprising:

means for restoring said saved register contents to said register from said save area.

**18**. The apparatus of claim **14** in which said program instruction specifies a beginning address of a save area and an offset from said beginning address, said restoring means comprising:

means for determining said storage location by adding the offset specified by said program instruction to the beginning address of the save area specified by said program instruction.

**19**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, apparatus for operating said processor to restore a previously saved program context, comprising:

means for decoding a program instruction specifying a register selected from said set of registers, said register pointing to a save area containing a saved program status word and saved register contents, and

means response to said decoding means for executing said instruction, said executing means comprising:

means for accessing said save area using the contents of the register specified by said program instruction; and

means for restoring said program status word and said register from the saved program status word and saved register contents contained in said save area to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

**20**. The apparatus of claim **19** in which the specified register specifies a base address, said program instruction also specifying a displacement from said base address, said accessing means comprising:

means for adding the specified displacement to the base address contained in said specified register.

**21**. The apparatus of claim **19** in which said register specifies a beginning address of said save area, said program instruction also specifying an offset from said beginning address, said accessing means comprising:

means for adding the offset specified by said program instruction to the beginning address of the save area specified by said register.

**22**. The apparatus of claim **19** in which said program executes in either a problem state having relatively restricted authority or a supervisor state having relatively unrestricted authority, said restoring means being operative for a program executing in said problem state and restoring only those fields of the current program status word that are alterable by a program executing in said problem state.

**23**. In an information handling system in which execution of a program of instructions by a processor is controlled by a program status word and by a set of registers defining a program context, said program status word containing an instruction address, apparatus for restoring a previously saved program status word and saved register contents defining a previous program context, comprising:

means for loading the address of a save area containing said previously saved program context into a specified one of said registers;

means for restoring the contents of said registers other than said specified register from said save area using a first instruction; and

means for restoring the contents of said program status word and said specified register from said save area using a second instruction to resume execution at the instruction address contained in said saved program status word with the program context defined by said saved program status word and saved register contents.

* * * * *

US006801993B2

(12) **United States Patent**
**Plambeck**

(10) **Patent No.: US 6,801,993 B2**
(45) **Date of Patent: Oct. 5, 2004**

(54) **TABLE OFFSET FOR SHORTENING TRANSLATION TABLES FROM THEIR BEGINNINGS**

(75) Inventor: **Kenneth E. Plambeck**, Poughkeepsie, NY (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 258 days.

(21) Appl. No.: **09/967,178**

(22) Filed: **Sep. 28, 2001**

(65) **Prior Publication Data**

US 2003/0074541 A1 Apr. 17, 2003

(51) **Int. Cl.**[7] ..... **G06F 12/00**
(52) **U.S. Cl.** ..... **711/206**; 711/6; 711/220
(58) **Field of Search** ..... 711/6, 203, 213, 711/217, 220, 221

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 4,569,018 | A * | 2/1986 | Hummel et al. | 711/207 |
| 4,802,084 | A * | 1/1989 | Ikegaya et al. | 711/6 |
| 4,899,275 | A * | 2/1990 | Sachs et al. | 711/3 |
| 4,945,480 | A * | 7/1990 | Clark et al. | 711/6 |
| 5,465,337 | A | 11/1995 | Kong | 711/207 |
| 5,530,820 | A * | 6/1996 | Onodera | 709/1 |
| 5,765,207 | A * | 6/1998 | Curran | 711/203 |
| 6,014,733 | A | 1/2000 | Bennett | 711/216 |
| 6,560,687 | B1 * | 5/2003 | Tsai et al. | 711/202 |

OTHER PUBLICATIONS

Microsoft Press Computer Dictionary, 3rd Edition, 1997, p. 339.*

"z/Architecture—Principles of Operation," IBM Publication No. SA22–7832–00 (Dec. 2000).

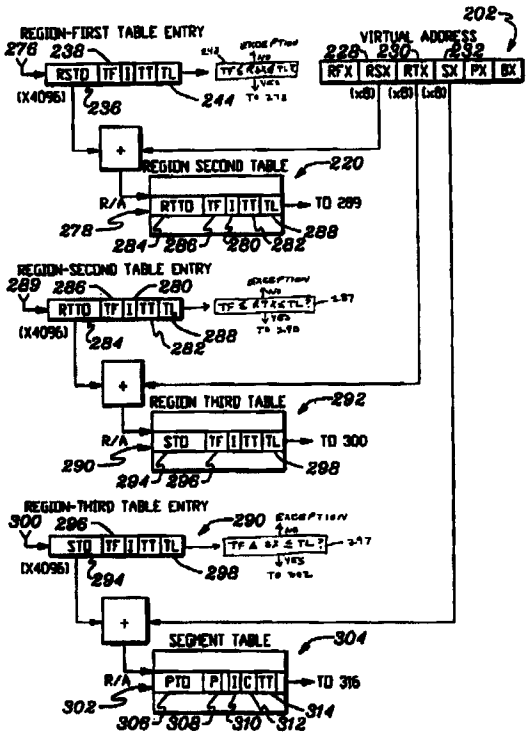
* cited by examiner

*Primary Examiner*—Nasser Moazzami
*Assistant Examiner*—Christian P. Chace
(74) *Attorney, Agent, or Firm*—Floyd A. Gonzalez, Esq.; Wayne F. Reinke, Esq.; Heslin Rothenberg Farley & Mesiti, P.C.

(57) **ABSTRACT**

A virtual address is translated to a real address using one or more tables at varying levels. An entry of a table is indexed based in part on a table origin and a table offset. The virtual address includes one or more indexes corresponding to the one or more varying level tables. A table is addressed as a function of the table origin and the corresponding index in the virtual address. The table offset indicates the actual beginning of the table from the origin.

**15 Claims, 4 Drawing Sheets**



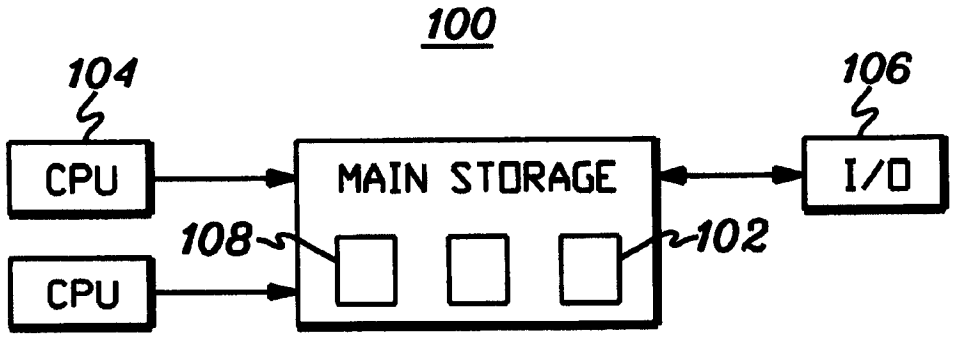



fig. 1

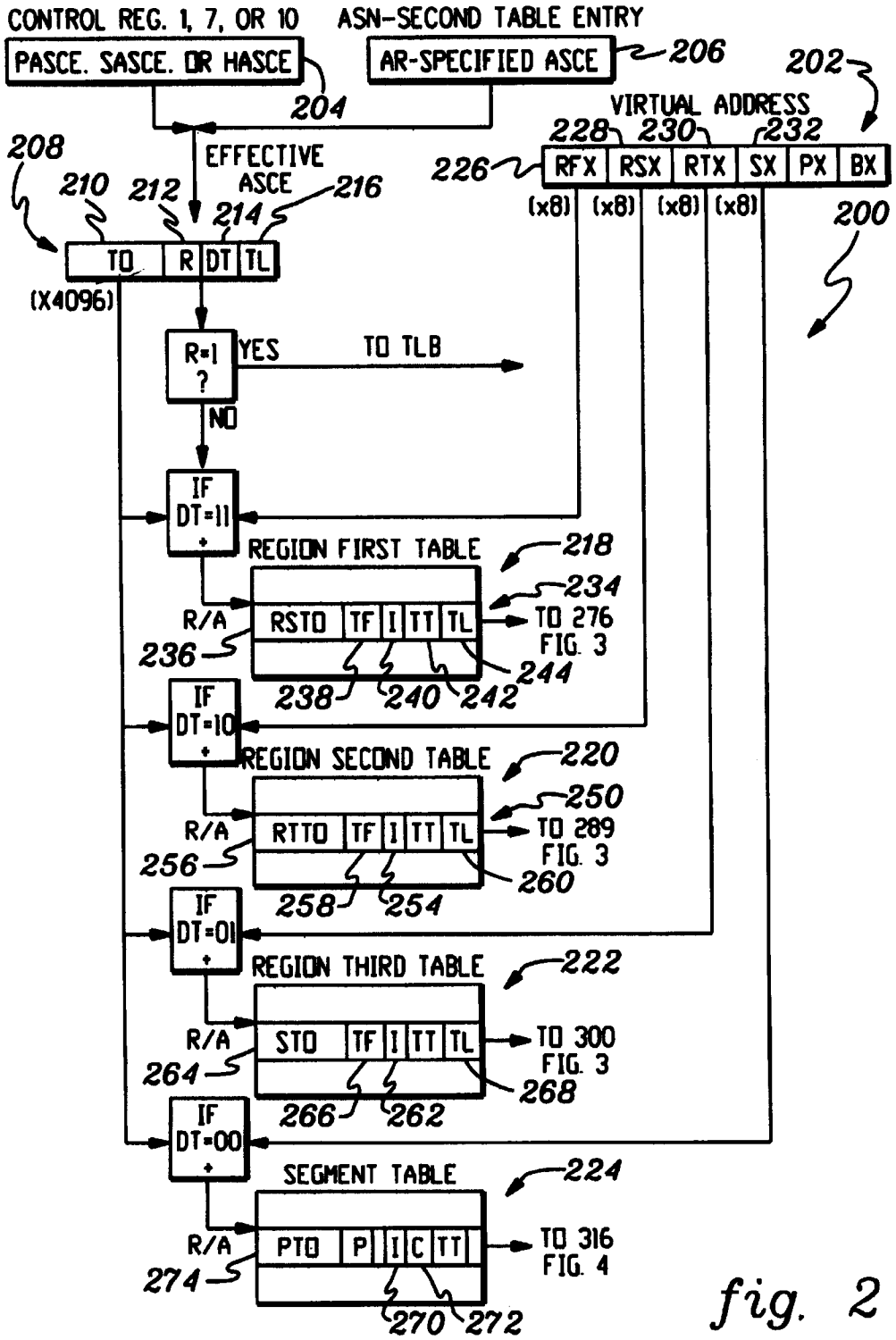

CONTROL REG. 1, 7, OR 10
PASCE. SASCE. OR HASCE
204
ASN-SECOND TABLE ENTRY
AR-SPECIFIED ASCE
206
202
VIRTUAL ADDRESS
228
230
232
226
RFX
RSX
RTX
SX
PX
BX
(x8)
(x8)
(x8)
(x8)
200
208
210
212
EFFECTIVE ASCE
214
216
TO
R
DT
TL
(X4096)
R=1 ?
YES
TO TLB
NO
IF DT=11
REGION FIRST TABLE
218
234
R/A
RSTO
TF
I
TT
TL
TO 276 FIG. 3
236
238
240
242
244
IF DT=10
REGION SECOND TABLE
220
250
R/A
RTTO
TF
I
TT
TL
TO 289 FIG. 3
256
258
254
260
IF DT=01
REGION THIRD TABLE
222
R/A
STO
TF
I
TT
TL
TO 300 FIG. 3
264
266
262
268
IF DT=00
SEGMENT TABLE
224
R/A
PTO
P
I
C
TT
TO 316 FIG. 4
274
270
272

fig. 2

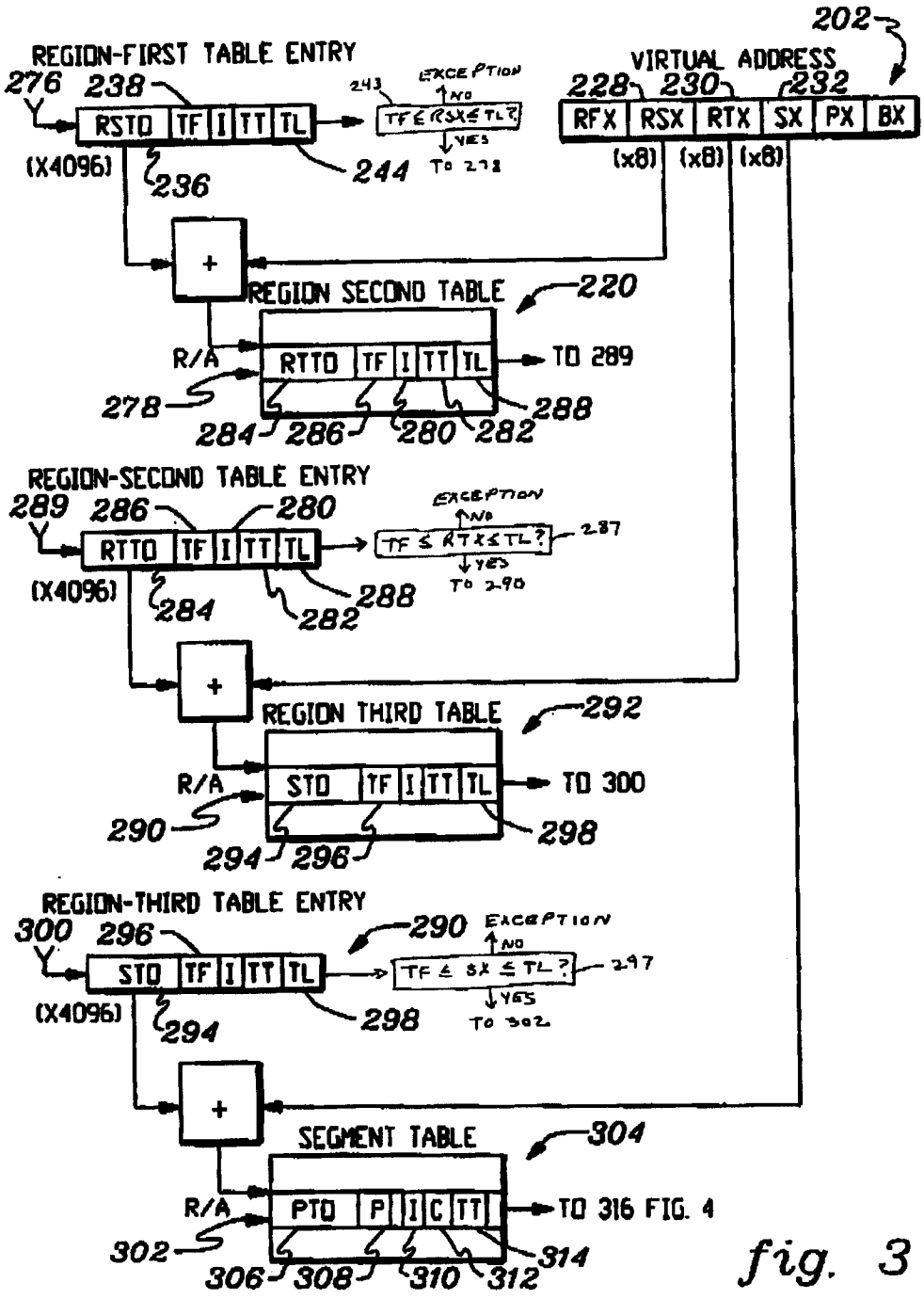

202
VIRTUAL ADDRESS
228 230 232
RFX RSX RTX SX PX BX
(x8) (x8) (x8)
REGION-FIRST TABLE ENTRY
276 238
RSTO TF I TT TL
(X4096)
236
243
EXCEPTION
NO
TF ≤ RSX ≤ TL?
YES
To 278
244
REGION SECOND TABLE
220
R/A
RTTO TF I TT TL
TO 289
278
284 286 280 282 288
REGION-SECOND TABLE ENTRY
289 286 280
RTTO TF I TT TL
(X4096)
284 282 288
EXCEPTION
NO
TF ≤ RTX ≤ TL?
287
YES
To 290
REGION THIRD TABLE
292
R/A
STO TF I TT TL
TO 300
290
294 296 298
REGION-THIRD TABLE ENTRY
300 296
STO TF I TT TL
(X4096)
294 298
290
EXCEPTION
NO
TF ≤ SX ≤ TL?
297
YES
To 302
SEGMENT TABLE
304
R/A
PTO P I C TT
TO 316 FIG. 4
302
306 308 310 312 314

fig. 3

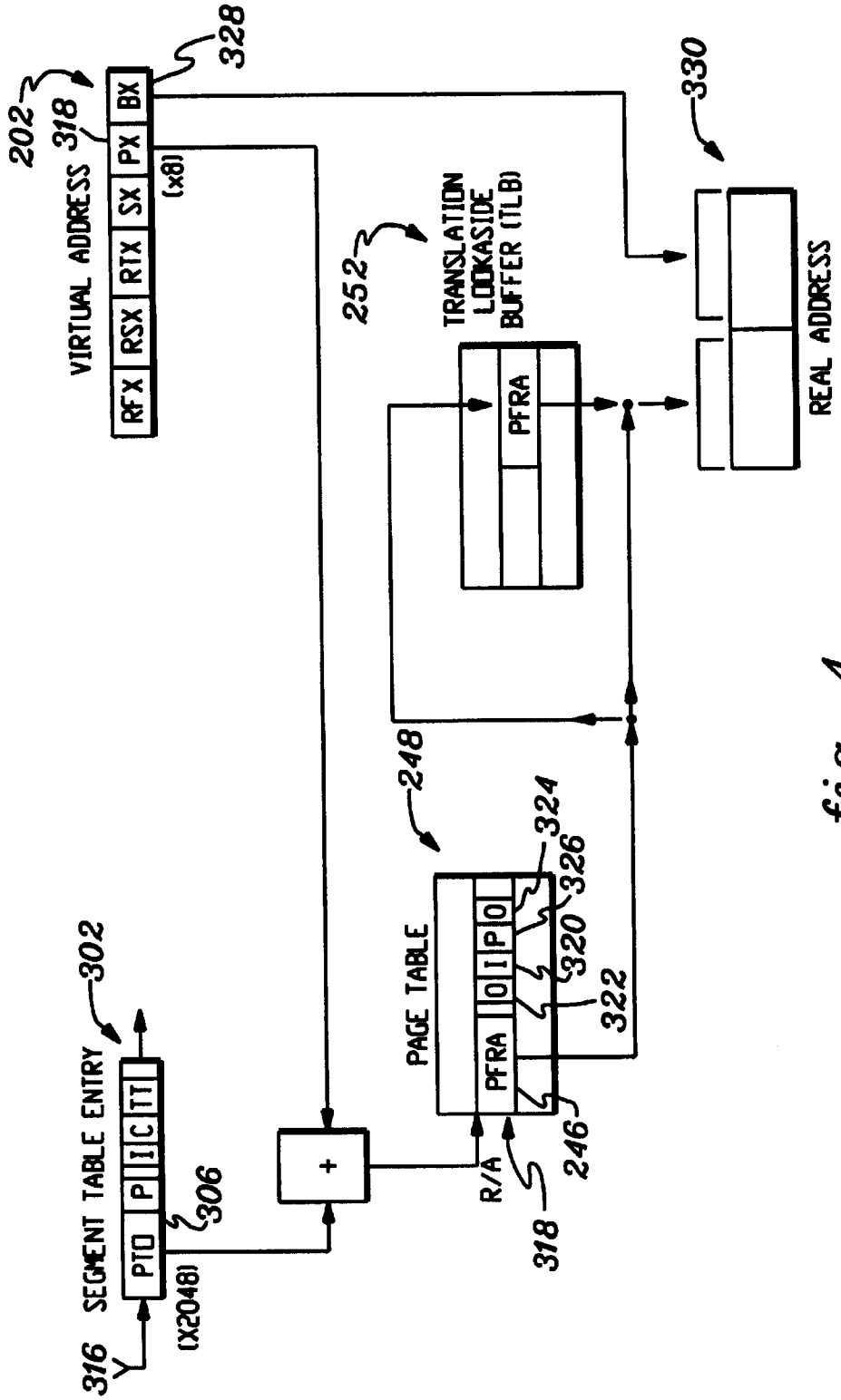

202
318
VIRTUAL ADDRESS
RFX
RSX
RTX
SX
PX
BX
(x8)
328
330
252
TRANSLATION LOOKASIDE BUFFER (TLB)
PFRA
REAL ADDRESS
248
324
326
320
322
PAGE TABLE
PFRA
I
P
246
R/A
318
302
SEGMENT TABLE ENTRY
PTO
I
C
TT
306
(X2048)
316
Y

fig. 4

# TABLE OFFSET FOR SHORTENING TRANSLATION TABLES FROM THEIR BEGINNINGS

## BACKGROUND OF THE INVENTION

1. Technical Field

The present invention generally relates to translation of virtual addresses to real addresses in a data processing system. More particularly, the present invention relates to the use of a table offset as part of the translation of a virtual address to a real address in a data processing system.

2. Background Informatiom

Data processing systems which use virtual addressing in multiple virtual address spaces are well known. Many data processing systems include, for example, a central processing unit (CPU) and a main storage. The CPU contains the sequencing and processing facilities for instruction execution, interruption action, timing functions, initial program loading and other machine related functions. The main storage is directly addressable and provides for high-speed processing of data by the CPU. The main storage may be either physically integrated with the CPU or constructed in stand-alone units.

In general, address spaces reside in main storage wherein an address space is a consecutive sequence of integer numbers (or virtual addresses), together with the specific transformation parameters which allow each number to be associated with a byte location in storage. The sequence starts at zero and proceeds left to right.

When a virtual address is used by a CPU to access main storage, it is first converted, by means of dynamic address translation (DAT), to a real address, and then, by means of prefixing, to an absolute address. DAT uses various levels of tables as transformation parameters. The designation (in the past, including origin and length) of a table is found for use by DAT in a control register or as specified by an access register.

DAT uses, at different times, the segment-table designations in different control registers or specified by the access registers. The choice is determined by the translation mode specified in the current program-status word (PSW). Four translation modes are available: primary-space mode, secondary-space mode, access-register mode (AR-mode), and home-space mode. Different address spaces are addressable depending on the translation mode.

Dynamic address translation (DAT) translates a virtual address of a computer system to a real address by means of translation tables. The bit string comprising a virtual address is divided, from left to right, into one or more table indexes and one byte index. The leftmost table index is multiplied by a table width and added to a predetermined table origin to form the address of an entry in the designated table. The next table index is multiplied by a table width and added to a table origin obtained from the entry in the first table to form the address of an entry in a second table. This process continues until all table indexes have been processed. The entry in the last table contains, instead of another table origin, a real address that is substituted for the concatenation of table indexes and concatenated with the byte index of the virtual address to form the real address resulting from the translation.

It has been the practice to include in the designation of the highest-level table and in each table entry that designates another table a field indicating the length of the designated table, at least when the table can be of significant size. A table-length field is a bit string of n bits. The leftmost n bits of an index are compared to the table-length bits for the corresponding table, and, if the value of the index bits is greater than the value of the table-length bits, the index is considered invalid and an exception is recognized (an interruption occurs) instead of proceeding with the translation. The table length has the advantage of saving storage that would be occupied by the unneeded end of a table.

However, it is sometimes the case that an address space is sparsely populated. One example is where an identifier of an object is used to form the address representation of that object in an address space. For example, if an object has an eight-character random name, the name could be used to form the address representation. It would be helpful in such situations to know where the necessary part of the designated table actually begins in order to save the storage that would otherwise be occupied by the unneeded beginning of the table.

Thus, a need exists for a way to indicate the actual beginning of a table in a virtual-to-real address translation.

## SUMMARY OF THE INVENTION

Briefly, the present invention satisfies the need for a way to indicate the actual beginning of a table in a virtual-to-real address translation by providing a table offset.

The table offset field saves the storage at the beginning of the table that would otherwise be occupied by the table.

In accordance with the above, it is an object of the present invention to indicate an actual beginning of a table in the translation of a virtual address to a real address.

The present invention provides, in a first aspect, a method of translating a virtual address to a real address. The method comprises indexing into an entry of a first table based on a table origin and a table offset.

Systems and program products corresponding to the method of the first aspect are also provided in second and third aspects of the invention, respectively.

These, and other objects, features and advantages of this invention will become apparent from the following detailed description of the various aspects of the invention taken in conjunction with the accompanying drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of one example of a system in accordance with the present invention.

FIGS. **2–4** are flow diagrams of one example of a virtual-to-real address translation in accordance with the present invention.

## DETAILED DESCRIPTION OF THE INVENTION

As shown in FIG. **1**, system **100** includes, for instance, a main storage **102**, one or more central processing units (CPUs) **104** and one or more input/output devices **106**.

In general, input devices **106** are used to load data and/or programs into main storage **102**, and central processing units **104** are used to access the stored program or data from main storage. As previously described, main storage **102** includes one or more address spaces **108**, wherein an address space is a consecutive sequence of integer numbers (or virtual addresses), together with the specific transformation parameters which allow each number to be associated with a byte location in storage. Typically, an entire virtual address space

108 is not resident within main storage. Instead, only that portion associated with a program or data being accessed or used by one or more of the processors is resident within the main storage.

The above-described computing environment and/or computing units are only offered as examples. The present invention can be incorporated and used with many types of computing units, computers, processors, nodes, systems, work stations and/or environments without departing from the spirit of the present invention. For example, the computing unit may be based on the UNIX architecture. Additionally, the present invention is relevant to servers and clients. Other types of computing environments can benefit from the present invention and are thus considered a part of the present invention.

The present invention will be described with reference to an example translation shown graphically in FIGS. **2–4** via flow diagram **200** and taken from IBM's z/Architecture, Principles of Operation, SA22-7832-00 (December 2000)—which-is herein incorporated by. In the z/Architecture translation of a virtual address **202** is controlled by the DAT-mode bit and address-space-control bits in the PSW and by the address-space-control elements (ASCEs) **204** in control registers (not shown) and as specified by the access registers **206**. When the ASCE used in a translation is a region-first-table designation, the translation is perfonned by means of a region first table, region second table, region third table, segment table, and page table, all of which reside in real or absolute storage, with the "or" in this construction meaning it is unpredictable whether prefixing is applied. When the ASCE is a lower-level type of table designation (region-second-table designation, region-third-table designation, or segment-table designation), the translation is performed by means of only the table levels beginning with the designated level, and the virtual-address bits that would, if nonzero, require use of a higher level or levels of table must be all zeros; otherwise, an ASCE-type exception is recognized. When the ASCE is a real-space designation, the virtual address is treated as a real address, and table entries in real or absolute storage are not used.

The address-space-control element (ASCE) used for a particular address translation is called the effective ASCE **208**. The effective ASCE is comprised of a table origin **210**, a real-space control **212**, a designation type **214** and a table length **216**. Accordingly, when a primary virtual address is translated, the contents of one control register are used as the effective ASCE. Similarly, for a secondary virtual address, the contents of another control register are used; for an AR-specified virtual address, the ASCE specified by the access register is used; and for a home virtual address, the contents of still another control register are used.

Although in this example the effective ASCE does not contain a table offset for the indicated table, it will be understood that it could. A table offset is less important for the table designated by the effective ASCE since there is only one such table. In contrast, in this example, there may be 2K (2,048) next-lower-level tables and, for each of them, another 2K next-lower-level tables, and so forth. Thus, in this example, if the effective ASCE designates a region first table, there may be a total of 8 G (8,589,934,592) segment tables, each requiring potentially 16K (16,384) bytes of storage.

When the effective address-space-control element (ASCE) **208** contains a real-space control **212**, having the value zero, the ASCE is a region-table or segment-table designation. When the real-space control is one, the ASCE is a real-space designation.

When the real-space control is zero, the designation-type **214** in the effective address-space-control element (ASCE), specifies the table-designation type of the ASCE. Depending on the type, some number of leftmost bits of the virtual address **202** being translated must be zeros; otherwise, an ASCE-type exception is recognized. For each possible value of the table-designation type, the indexes in the virtual-address required to be zeros are as follows in Table I:

TABLE I

| DT 214 | Designation Type | Virtual-Address Portion (s) Required to Be Zeros |
|---|---|---|
| 11 | Region-first-table 218 | None |
| 10 | Region-second-table 220 | Region first index 226 |
| 01 | Region-third-table 222 | Region first and second indexes 226, 228 |
| 00 | Segment-table 225 | Region first, second and third indexes 226, 228, 230 |

The designation-type **214** of the effective address-space-control element (ASCE) specifies both the table-designation type of the ASCE and the portion of the virtual address that is to be translated by means of the designated table, as follows in Table II:

TABLE II

| DT 214 | Designation Type | Virtual-Address Portion Translated by the Table |
|---|---|---|
| 11 | Region-first-table 218 | Region first index 226 |
| 10 | Region-second-table 220 | Region second index 228 |
| 01 | Region-third-table 222 | Region third index 230 |
| 00 | Segment-table 224 | Segment index 232 |

In addition to the region and segment indexes, the virtual address also includes a page-table index (PX) and a byte index (BX). The byte index is eventually concatenated with the page frame real address **246** of the page table **248** to obtain the real address.

When the designation type has the value **11** binary, the region-first-index portion **226** of the virtual address, in conjunction with the table origin **210** contained in the ASCE, is used to select an entry **234** from the region first table. The entry is comprised of a region-second-table origin **236**, a table offset **238**, an invalid bit **240**, a table type **242**, and a table length **244**. The table offset indicates where the region-second table actually begins relative to the RSTO. The inclusion of the table offset saves the space between RSTO and the actual beginning that would otherwise be occupied by the region-second table. The table length indicates the length of the region-second table taken from the RSTO of the table.

The 64-bit address of the region-first-table entry **234** in real or absolute storage ("R/A" in FIGS. **2–4**), is obtained by appending 12 zeros to the right of bits **0–51** of the region-first-table designation (i.e., conceptually multiplying by 4096) and adding the region first index with three rightmost and 50 leftmost zeros appended (i.e., conceptually multiply-

ing the region first index by 8, the width of a table entry). When a carry out of bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{64}-1$ to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the region-first-table-lookup process, the first two bits of the region first index in the virtual address are compared against the table length **216** of the region-first-table designation, to establish whether the addressed entry is within the region first table. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-first-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-first-table entry in the translation-lookaside buffer **252** is used in the translation.

All eight bytes of the region-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

Invalid bit **240** of the entry fetched from the region first table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-first-translation exception is recognized.

A translation-specification exception is recognized if the table-type **242** in the region-first-table entry does not have the same value as the designation type **214** of the ASCE.

When no exceptions are recognized in the process of region-first-table lookup, the entry fetched from the region first table designates the origin and specifies the offset and length of the corresponding region second table.

When the designation type **214** of the ASCE has the value **10** binary, the region-second-index **228** portion of the virtual address **202**, in conjunction with the table origin **210** contained in the ASCE, is used to select an entry from the region second table **220**. The first two bits of the region second index **228** are compared against the table length **216** in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-second-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-second-table entry in the translation-lookaside buffer **252** is used in the translation. The region-second-table-lookup process is otherwise the same as the region-first-table-lookup process, except that a region-second-translation exception is recognized if the invalid bit **254** is one in the region-second-table entry. When no exceptions are recognized, the entry fetched from the region second table designates the origin **256** and specifies the offset **258** and length **260** of the corresponding region third table.

When the designation type **214** of the ASCE has the value 01 binary, the region-third-index **230** portion of the virtual address, in conjunction with the table origin **210** contained in the ASCE, is used to select an entry from the region third table **222**. The first two bits of the region third index **230** are compared against the table length **216** in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-third-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-third-table entry in the translation-lookaside buffer **252** is used in the translation. The region-third-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if the invalid bit **262** is one in the region-third-table entry. When no exceptions are recognized, the entry fetched from the region third table designates the origin **264** and specifies the offset **266** and length **268** of the corresponding segment table.

When the designation type **214** of the ASCE has the value 00 binary, the segment-index portion **232** of the virtual address **202**, in conjunction with the table origin **218** contained in the ASCE, is used to select an entry from the segment table **224**. The first two bits of the segment index **232** are compared against the table length **216** in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a segment-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a segment-table entry in the translation-lookaside buffer **252** is used in the translation. A segment-translation exception is recognized if the invalid bit **270** is one in the segment-table entry. The segment-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if the invalid bit **270** is one in the segment-table entry. When no exceptions are recognized, the entry fetched from the segment table designates the origin **274** of the corresponding page table.

When the effective address-space-control element (ASCE) is a region-table designation, a region-table entry is selected as described above. The contents of the selected entry and the next index portion of the virtual address are used to select an entry in the next-lower-level table, which may be another region table or a segment table.

When the table entry selected by means of the effective ASCE is a region-first-table entry, the region-second-index portion of the virtual address, in conjunction with the region-second-table origin contained in the region-first-table entry, is used to select an entry from the region second table.

As shown at **276** in FIG. **3**, the 64-bit address of the region second table entry **278** in real or absolute storage is obtained by appending 12 zeros to the right of bits **0–51** of the region first table entry and adding the region second index with three rightmost and 50 leftmost zeros appended. In other words, the region-second-table origin **236** with appended zeros and the region-second-table index **228** with appended zeros are added together. When a carry out of bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{64}-1$ to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the region-second-table-lookup process, the first two bits of the region second index **228** are compared **243** against the table offset **238** of the region-first-table entry, and against the table length **244** of the region-first-table entry, to establish whether the addressed entry is within the region second table. If the value in the table-offset field is greater than the value in the corresponding bit positions of the virtual address, or if the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-second-translation exception is recognized.

All eight bytes of the region-second-table entry **278** appear to be fetched concurrently as observed by other

CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

An invalid bit **280** of the entry fetched from the region second table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-second-translation exception is recognized.

A translation-specification exception is recognized if the table type **282** in the region-second-table entry does not have a value that is one less than the value of the table type in the next-higher-level table, i.e., table type **242**.

When no exceptions are recognized in the process of region-second-table lookup, the entry **278** fetched from the region second table designates the beginning **284** and specifies the offset **286** and length **288** of the corresponding region third table.

As shown at **289** in FIG. **3**, when the table entry selected by means of the effective ASCE **208** is a region-second-table entry **278**, or if a region-second-table entry has been selected by means of the contents of a region-first-table entry **234**, the region-third-index portion **230** of the virtual address **202**, in conjunction with the region-third-table origin **284** contained in the region-second-table entry, is used to select an entry **290** from the region third table **292**. The first two bits of the region third index **230** are compared **287** against the table offset **286** and table length **288** in the region-second-table entry. A region-third-translation exception is recognized if the table offset is greater than the first two bits, if the table length is less than the first two bits, or if the invalid bit **280** is one in the region-third-table entry. The region-third-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if the invalid bit **262** is one in the region-third-table entry. When no exceptions are recognized, the entry **290** fetched from the region third table **292** designates the origin **294** and specifies the offset **296** and length **298** of the corresponding segment table.

As shown at **300** in FIG. **3**, when the table entry selected by means of the effective ASCE is a region-third-table entry, or if a region-third-table entry has been selected by means of the contents of a region-second-table entry, the segment-index portion **232** of the virtual address, in conjunction with the segment-table origin **294** contained in the region-third-table entry **290**, is used to select an entry **302** from the segment table **304**. Entry **302** comprises a page table origin **306**, a page protection bit **308**, an invalid bit **310**, a common segment bit **312**, and a table type field **314**.

The first two bits of the segment index **232** are compared **297** against the table offset **296** and table length **298** in the region-third-table entry. A segment-translation exception is recognized if the table offset is greater than the first two bits, if the table length is less than the first two bits, or if the invalid bit **310** is one in the segment-table entry. A translation-specification exception is recognized if (1) the page protection bit **308** in the entry is one and (2) the common-segment bit **312** in the entry fetched from the segment table is one. The segment-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if the invalid bit **270** is one in the segment-table entry. When no exceptions are recognized, the entry fetched from the segment table designates the origin **306** of the corresponding page table (see **316** in FIG. **4**)

As shown in FIG. **4**, the page-index portion **318** of the virtual address **202**, in conjunction with the page-table origin **306** contained in the segment-table entry **302**, is used to select an entry **318** from the page table **248**.

The 64-bit address of the page-table entry **318** in real or absolute storage is obtained by appending 11 zeros to the right of the page-table origin **306** and adding the page index **318**, with three rightmost and 53 leftmost zeros appended. A carry out of bit position 0 cannot occur. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All eight bytes of the page-table entry appear to be fetched concurrently as observed by other CPUS. The fetch access is not subject to protection. When the storage address generated for fetching the page-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

The entry **318** fetched from the page table **248** indicates the availability of the page and contains the leftmost bits of the page-frame real address **246**. The page-invalid bit **320** is inspected to establish whether the corresponding page is available. If this bit is one, a page-translation exception is recognized. If portions **322** and **324** of the page table entry contain a one, a translation-specification exception is recognized. If the page-protection bit **326** is one either in the segment-table entry used in the translation or in the page-table entry, and the storage reference for which the translation is being performed is a store, a protection exception is recognized.

When the effective address-space-control element (ASCE) is a region-table designation or a segment-table designation and no exceptions in the translation process are encountered, the page-frame real address **246** is obtained from the page-table entry. When the ASCE is a real-space designation (i.e., bit **212** in the effective ASCE **208** is one), the virtual address except for the byte index is used as a page-frame real address. In either case, the page-frame real address **246** and the byte-index portion **328** of the virtual address **202** are concatenated, with the page-frame real address forming the leftmost part. The result is the real storage address **330** which corresponds to the virtual address. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

To enhance performance, the dynamic-address-translation mechanism normally is implemented such that some of the information specified in the region tables, segment tables, and page tables is maintained in a special buffer, referred to as the translation-lookaside buffer (TLB) **252**. The CPU necessarily refers to a DAT-table entry in real or absolute storage only for the initial access to that entry. This information may be placed in the TLB, and subsequent translations may be performed by using the information in the TLB. For consistency of operation, the virtual-equals-real translation specified by a real-space designation also may be performed by using information in the TLB. The presence of the TLB affects the translation process to the extent that (1) a modification of the contents of a table entry in real or absolute storage does not necessarily have an immediate effect, if any, on the translation, (2) a region-first-table origin, region-second-table origin, region-third-table origin, segment-table origin, or real-space token origin in an address-space-control element (ASCE) may select a TLB

entry that was formed by means of an ASCE containing an origin of the same value even when the two origins are of different types, and (3) the comparison against the table length in an address-space-control element may be omitted if a TLB equivalent of the designated table entry is used. In a multiple-CPU configuration, each CPU has its own TLB.

Entries within the TLB are not explicitly addressable by the program.

Information is not necessarily retained in the TLB under all conditions for which such retention is permissible. Furthermore, information in the TLB may be cleared under conditions additional to those for which clearing is mandatory.

The present invention can be included in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. The media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities of the present invention. The article of manufacture can be included as a part of a computer system or sold separately.

Additionally, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform the capabilities of the present invention can be provided.

The flow diagrams depicted herein are just exemplary. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

While several aspects of the present invention have been described and depicted herein, alternative aspects may be effected by those skilled in the art to accomplish the same objectives. Accordingly, it is intended by the appended claims to cover all such alternative aspects as fall within the true spirit and scope of the invention.

What is claimed is:

**1**. A method of translating a virtual address to a real address, the method comprising:

indexing into an entry of a first table based on a translation table origin, wherein the entry of the first table comprises a second table origin and a second table offset; and

indexing into an entry of a second table based on the second table origin and the second table offset, wherein indexing into an entry of the second table comprises:

comparing an indicator in the virtual address to the second table offset; and

proceeding with the indexing if the indicator is equal to or greater than the second table offset.

**2**. The method claim **1**, wherein the entry of the first table also comprises a table length indicator, wherein the comparing also comprises comparing the indicator to the table length indicator, and wherein the proceeding comprises proceeding with the indexing if the indicator is equal to or greater than the table offset and less than or equal to the table length indicator.

**3**. The method claim **1**, wherein the second table is a next-lower-level table than the first table.

**4**. The method of claim **1**, wherein the virtual address comprises at least two table indexes, each table corresponding to one of the at least two table indexes, and wherein the indexing into the entry of the second table is also based on an index of the at least two table indexes corresponding to the second table.

**5**. The method of claim **1**, wherein the virtual address comprises at least one table index, and wherein the indexing into the entry of the first table is also based on the at least one table index.

**6**. A system for translating a virtual address to a real address, comprising:

means for indexing into an entry of a first table based on a translation table origin, wherein the entry of the first table comprises a second table origin and a second table offset; and

means for indexing into an entry of a second table based on the second table origin and the second table offset, wherein the means for indexing into an entry of the second table comprises:

means for comparing an indicator in the virtual address to the second table offset; and

means for proceeding with the indexing if the indicator is equal to or greater than the second table offset.

**7**. The system of claim **6**, wherein the entry of the first table also comprises a table length indicator, wherein the means for comparing also comprises means for comparing the indicator to the table length indicator, and wherein the means for proceeding comprises means for proceeding with the indexing if the indicator is equal to or greater than the table offet and less than or equal to the table length indicator.

**8**. The system of claim **6**, wherein the second table is a next-lower-level table than the first table.

**9**. The system of claim **6**, wherein the virtual address comprises at least one table index, and wherein the means for indexing into the entry of the first table is also based on the at least one table index.

**10**. The system of claim **6**, wherein the virtual address comprises at least two table indexes, each table corresponding to one of the at least two table indexes, and wherein the means for indexing into the entry of the second table is also based on an index of the at least two table indexes corresponding to the second table.

**11**. At least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform a method of translating a virtual address to a real address, the method comprising:

indexing into an entry of a first table based on a translation table origin and a wherein the entry of the first table comprises a second table origin and a second table offset; and

indexing into an entry of a second table based on the second table origin and the second table offset, wherein indexing into an entry of the second table comprises:

comparing an indicator in the virtual address to the second table offset; and

proceeding with the indexing if the indicator is equal to or greater than the second table offset.

**12**. The at least one program storage device of claim **11**, wherein the entry of the first table also comprises a table length indicator, wherein the comparing also comprises comparing the indicator to the table length indicator, and wherein the proceeding comprises proceeding with the indexing if the indicator is equal to or greater than the table offset and less than or equal to the table length indicator.

**13**. The at least one program storage device of claim **11**, wherein the second table is a next-lower-level table than the first table.

**14**. The at least one program storage device of claim **11**, wherein the virtual address comprises at least two table indexes, each table corresponding to one of the at least two table indexes, and wherein the indexing into the entry of the

second table is also based on an index of at least two table indexes corresponding to the second table.

15. The at least one program storage device of claim 11, wherein the virtual address comprises at least one table index, and wherein the indexing into the entry of the first table is also based on the at least one table index.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 6,801,993 B2
DATED : October 5, 2004
INVENTOR(S) : Plambeck

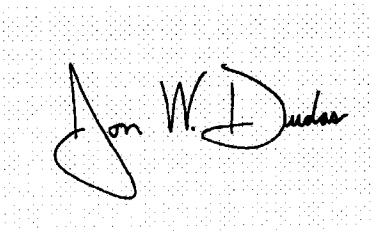


It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 3,
Lines 19 and 20, delete "-which-is herein incorporated by." after "(December 2000)" and insert -- . -- after the ")"

Signed and Sealed this

Seventh Day of December, 2004



JON W. DUDAS
*Director of the United States Patent and Trademark Office*